Computer Science Department

TECHNICAL REPORT

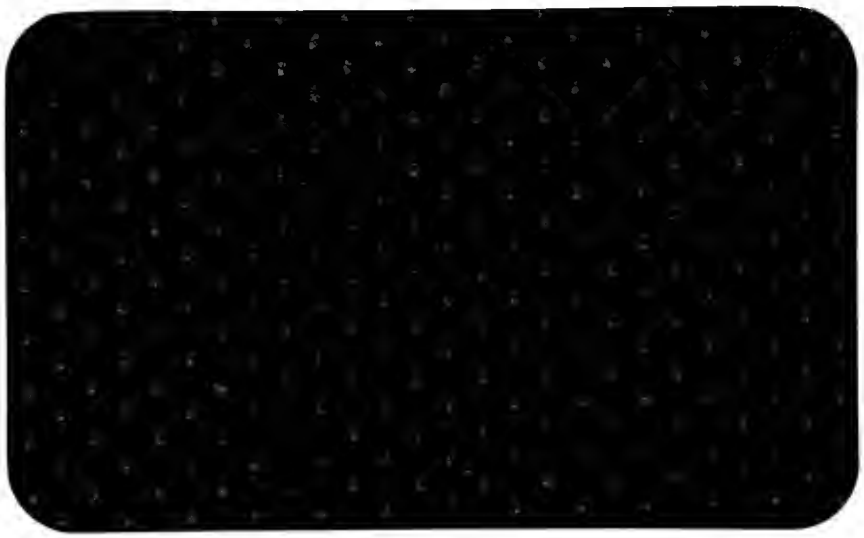
INTERNAL, EXTERNAL, AND PRAGMATIC INFLUENCES: TECHNICAL PERSPECTIVES IN THE DEVELOPMENT OF PROGRAMMING LANGUAGES

by

JACOB T. SCHWARTZ

July 1980

Report No. 020



Technical Survey No. 20

INTERNAL, EXTERNAL, AND PRAGMATIC INFLUENCES: TECHNICAL PERSPECTIVES IN THE DEVELOPMENT OF PROGRAMMING LANGUAGES

by

JACOB T. SCHWARTZ

July 1980
Report No. 020

This report was prepared under Grant No. N00014-78-C-0639 from the Office of Naval Research.

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 020	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: Internal, External, and Pragmatic Influences: Technical Perspectives in the Development of Programming Languages	5. TYPE OF REPORT & PERIOD COVERED: Technical
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): Jacob T. Schwartz	8. CONTRACT OR GRANT NUMBER(s): N00014-78-C-0639
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, N.Y. 10012	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research	12. REPORT DATE: July 1980
	13. NUMBER OF PAGES: 72
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Department of the Navy, Arlington, Virginia 22217	15. SECURITY CLASS. *(of this report)*: unclassified
	15e. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

none

18. SUPPLEMENTARY NOTES

none

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

programming methodology
programming languages
high-level programming techniques
specification techniques

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This paper reviews key issues likely to shape the design and development of programs and programming languages as efficiency constraints relax and computational power increases. These issues are;

1) the use of the dictions of mathematics to define operations that can be used to achieve user-defined desires and expectations concerning an intended application;
2) techniques for designing and producing the expected results: the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

use of very-high-level programming languages for prototyping; the development of application-oriented languages; the design of applications using standard operations; and the development of a library of modules which handle substantial fragments of important applications;

3) a language's usefulness, i.e., its general nature and its user-oriented features.

Technical Perspectives in the Development of Programming Languages

1. Introduction. External and internal issues in program design.

Integrated circuit technology is evolving continuously and rapidly toward smaller elementary devices and denser, more complex functions on each silicon chip. This is bringing a greatly expanded computational power into being, with rapidly falling costs. The following stages of development can be anticipated:

(a) Presently, computers (like the DEC VAX) which provide large-machine performance (e.g. 1-million-instructions-per-second instruction rate) but at greatly reduced cost (e.g. $300,000 vs. $1.5 million or more) have become available.

(b) Over the next ten years, machines of this capability should become available at costs ranging down to $10,000 or less.

(c) Within two decades, these same capabilities should become available in home computers selling for approximately $1,000.

(d) Various supporting technologies, especially bulk storage technology (e.g. videodisc), speech generation, and low-cost printing are also developing rapidly.

The increased computational power that is coming into being is bound to impact programming technique significantly. A central possibility here is to *use expanding computational power in radical and aggressive fashion to alleviate the problems of software production*, generally recognized today as a main obstacle to expanded computer application. In order to assess the potential for development in this direction, we shall review various key issues likely to shape the design and development of programming languages as the efficiency constraints which have been all-dominating until now relax progressively. This review is organized around several main themes:

(i) Ingredients of two fundamental sorts enter into the composition of a program. Material of the first category serves to define user desires and expectations concerning an

intended application, for example the nature of expected input, and of output including output text and graphics, prompts and warnings, error diagnostics, etc. This material, which can in fact constitute the overwhelming bulk of a particular application, is motivated by *external* considerations having an intrinsically nonmathematical character. Material of a second, contrasting category serves to define the toolbox of operations which can be used to achieve the desired external behavior; this *internal* program material has a much more mathematical character. Depending on the relative weight of program material belonging to these two categories, a program may be called *externally* or *internally motivated*. In this sense, much of an optimizing compiler is internally motivated, whereas by far the greater part of a commercial report generation program is externally motivated.

We shall see that the dictions of mathematics serve very adequately to define the internally motivated portions of a program, and that closely related dictions can be used to describe the algorithms which allow mathematical operations to be realized with acceptable efficiency. Beyond this, other mathematical questions, having to do with verification of algorithm correctness and transformation of programs between equivalent forms having different levels of efficiency, attach naturally to the internally motivated aspects of programs.

(ii) No equally satisfactory tools for dealing with the much more varied externally motivated aspects of programs are available. However, we shall suggest a variety of techniques for making it easier to design and produce the externally motivated portions of a program. These include:

(ii.a) The use of very-high-level languages for system prototyping, so that users can be exposed to a running system version for preliminary reaction before a very large system development expenditure has been incurred. This would allow functional deficiencies to be caught while it was still possible to correct them relatively cheaply.

(ii.b) Development of application-oriented languages, which allow the characteristic problems of an area to be organized conveniently by making central notions of the area available as language primitives.

(ii.c) Systematic attempts to design applications using standardized operations rather than specially tailored operation sequences.

(ii.d) Development of a library of modules which handle substantial fragments of important applications, but which can also be used as building blocks for more complex applications. Typical modules might be: an on-line editor module, usable as a general-purpose command-input facility; a grammar-driven parser capable of generating useful standardized diagnostics for relatively free form input; standardized graphics modules for display output; etc.

Since various existing languages provide useful application-oriented features, we also consider the possibility of linking existing languages together into a multipurpose 'programming language environment'.

(iii) The rate at which the user of a language is able to progress in developing an application depends not only on the general nature of the language but also on its user-oriented features. In the final section of this report we review a variety of facilities, including portability aids, debugging aids, and program measurement facilities whose presence or absence can have a strong pragmatic effect on a language's usefulness.

2. The algorithmic side of programming.

A. Mathematics as a programming language.

The set-theoretic foundations of mathematics involve remarkably few primitive constructions. We can enumerate these as follows:

(a) The operators $\&$, $\vee$, $\rightarrow$, $\leftrightarrow$ of propositional calculus, and also the quantifiers $(\forall x)$, $(\exists x)$ of predicate calculus, together with all the standard rules for manipulating them, are available. It is also convenient to allow conditional terms and predicates of the form

$$\text{if } C_1 \text{ then } t_1 \text{ elseif } C_2 \text{ then } t_2 \ldots \text{ else } t_n .$$

(b) The equality relation $x = y$ and the membership relation $x \in y$ are available. Equality has all its standard properties, and in addition we have the

Axiom: $u = v \leftrightarrow (\forall x)(x \in u \leftrightarrow x \in v)$.

(c) We are allowed to write set formers $\{e: x_1,\ldots,x_n: C\}$, where e is any expression, C any predicate, and $x_1,\ldots,x_n$ any list of variables. This syntactic construct designates the set of all values that e can take on as $x_1,\ldots,x_n$ vary over all values satisfying the condition C. We therefore have the

Axiom: $z \in \{e: x_1,\ldots,x_n: C\} \leftrightarrow (\exists x_1 \ldots x_n)\ (z = e \ \&\ C)$.

The expression e and condition C are essentially arbitrary, except that certain technical restrictions, which prevent formation of 'paradoxical' sets (e.g. the set of all sets which are not members of themselves), must be respected. (We shall not discuss these restrictions here.)
It is convenient to use $\{x:C\}$ to abbreviate $\{x:x:C\}$.

(d) If f is any function (resp. predicate) symbol which has never been used before, and e is any expression (resp. predicate) whose only free variables are $x_1,\ldots,x_n$, we can introduce the equality

$f(x_1,\ldots,x_n) = e$ (resp. $f(x_1,\ldots,x_n) \leftrightarrow e$)

as a definition. (For emphasis, we will generally write such equalities as $f(x_1,\ldots,x_n)$ ::= e. As a syntactic convenience, we will also allow functions introduced in this way to be written as infix or prefix operators, or to be indicated by use of other convenient syntactic forms, e.g. special brackets.)

Many basic set-theoretic definitions are entirely straightforward:

$\{u\}$ ::= $\{y: y=u\}$	(singleton)
$\{u,v\}$ ::= $\{y: y=u \vee y=v\}$	(pair)
$\emptyset$ ::= $\{y: y \neq y\}$	(nullset)
0 ::= $\emptyset$, 1 ::= $\{0\}$, 2 ::= $\{0,1\}$	(zero, one, and two)
$\langle x,y\rangle$ ::= $\{\{0,\{x\}\},\{2,\{y\}\}\}$	(ordered pair)
$u \cup v$ ::= $\{y: y \in u \vee y \in v\}$	(union)
$u \cap v$ ::= $\{y: y \in u \& y \in v\}$	(intersection)
$u \setminus v$ ::= $\{y: y \in u \& y \notin v\}$	(difference)
$u \subseteq v$ ::$\leftrightarrow$ $(u \vee v = \emptyset)$	(inclusion)
pow(u) ::= $\{y: y \subseteq u\}$	(powerset)
$f\{u\}$ ::= $\{y: \langle u,y\rangle \in f\}$	(multivalued map application)
Rf ::= $\{y: x,y: \langle x,y\rangle \in f\}$	(range)
Df ::= $\{x: x,y: \langle x,y\rangle \in f\}$	(domain)
$f\|s$ ::= $\{\langle x,y\rangle: x,y: \langle x,y\rangle \in f \& x \in s\}$	(restriction of map to set)
$f[s]$ ::= $R(f\|s)$	(range of map on set)
f^{-1} ::= $\{\langle y,x\rangle: x,y: \langle x,y\rangle \in f\}$	(inverse mapping)
$f \circ g$ ::= $\{\langle x,z\rangle: x,y,z: \langle x,y\rangle \in f \& \langle y,z\rangle \in g\}$	(functional composition)
hd(u) ::= $\eta\eta\{x-\{0\}: x: x \in u \& 0 \in x\}$	(first component of ordered pair)
tl(u) ::= $\eta\eta\{x-\{2\}: x: x \in u \& 2 \in x\}$	(second component " ")
Un(s) ::= $\{x: x,y: x \in y \& y \in s\}$	(union set)

Under appropriate restrictions, definitions are allowed to be recursive, i.e. the function symbol appearing on the left of a definition can also appear on the right. Specifically,

let the definition be

(*) $$f(x_1,\ldots,x_n) ::= r \ .$$

Then there must exist some other, already defined function $a(x_1,\ldots,x_n)$, which we shall call the auxiliary function of the definition (*), such that every occurrence of f within r is part of a subexpression

$$\text{if } a(e_1,\ldots,e_n) \notin a(x_1,\ldots,x_n) \text{ then } \emptyset \text{ else } f(e_1,\ldots,e_n) \ .$$

Similarly, for recursive predicate definitions

$$P(x_1,\ldots,x_n) :\Leftrightarrow r \ ,$$

we insist that every occurrence of P within r be part of a predicate subterm

$$a(e_1,\ldots,e_n) \in a(x_1,\ldots,x_n) \rightarrow P(e_1,\ldots,e_n) \ .$$

It should be understood that only the initial form of a recursive definition is subject to these rules; this initial form can then be rewritten in any equivalent form for convenience where desired.

(e) For any predicate $P(x_1,\ldots,x_n,y)$ whose only free variables are $x_1,\ldots,x_n,y$ we can introduce a new function symbol f of n variables and a defining statement

$$(\forall x_1,\ldots,x_n)\ (P(x_1,\ldots,x_n,f(x_1,\ldots,x_n)) \Leftrightarrow (\exists y)\ P(x_1,\ldots,x_n,y)).$$

(f) The necessary assumption that there exists at least one infinite set is formulated as the

Axiom: $(\exists u)\ u \subseteq Un(u)\ \&\ u \neq \emptyset$ (axiom of infinity).

(g) A special 'choice' operator η which selects a (fixed) element from each nonnull set is available, together with the

Axiom: $\eta s \cap s = \emptyset\ \&\ (\eta s \in s \vee s = \emptyset)\ \&\ \eta\emptyset = \emptyset$ (axiom of choice).

As an indication of the astonishing power of this simple system, we will now review the basic definitions of various major areas of mathematics, beginning with

Integer arithmetic (including ordinal arithmetic).

The integers are encoded as $2 = \{0,1\}$, $3 = \{0,1,2\}$, etc., making it intuitively plain that every integer m has the properties: $n \supseteq Un(n)$ and $(\forall x \in n)(\forall y \in n)\ (x \in y \vee y \in x \vee x = y)$. This leads to the following definitions:

$Elo(s) :\Leftrightarrow (\forall x \in s)(\forall y \in s)\ (x \in y \vee y \in x \vee x = y)$

$Ord(s) :\Leftrightarrow s \supseteq Un(s) \ \&\ Elo(s)$ (s is an ordinal number)

$f(x) ::= \cap f\{x\}$ (image element)

$Carrier(f) ::= \{x: x \in Df \ \&\ f(x) \neq 0\}$

$Singlevalued(f) ::= (\forall x \in Df)\ f\{x\} = \{f(x)\}$ (singlevalued map)

$s \times t ::= \{\langle x,y\rangle: x \in s \ \&\ y \in t\}$ (Cartesian product)

$Maps(s,t) ::= \{f: f \subseteq s \times t \ \&\ Df = s \ \&\ Singlevalued(f)\}$ (maps from s to t)

$\#s ::= \cap\{x: Ord(x) \ \&\ (\exists f \in Maps(x,s))\ (Rf = s)\}$ (cardinality)

A set is a cardinal if $\#s = s$. The product, sum, and difference of cardinals are defined by:

$$s * t ::= \#(s \times t)$$
$$s + t ::= \#(\{0\} \times s \cup \{1\} \times t)$$
$$s - t ::= \#(s \setminus t)$$

If u_0 is any set having the property stated in the axiom of infinity, then we can put

$$Z_+ ::= \cap\{x: x \in \#pow(u_0) \ \&\ x = Un(x)\},$$

thus defining the set of all nonnegative integers.
Note that the manner in which we have defined the integers makes every nonnegative integer the set of all smaller nonnegative integers. We can then put

$$Seq(f) ::= Singlevalued(f) \ \&\ Df \in Z_+ \quad (f \text{ is a sequence}).$$

The successor of any cardinal (or ordinal) n is $n \cup \{n\}$ and its predecessor (if any) is $Un(n)$.

As an example of a recursive definition, we shall define the quantity comb(m,f) that would normally be written

$$m(\ldots(m(m(f(0),f(1)),f(2))\ldots f(\#f-1)),$$

i.e., the combination of all the components of the sequence f using the binary operation m. This has the definition

$$\text{comb}(m,f) ::= \text{if } \neg\text{Seq}(f) \vee \#f \in \{0,1\} \text{ then } f(0)$$
$$\text{else } m(\text{comb}(m, f|\text{Un}(\text{Df})), f(\text{Un}(\text{Df}))) .$$

Since $\text{Un}(n) \in n$ for each $n \in Z_+$, we have $\#(f|\text{Un}(\text{Df})) \in \#\text{Df}$ for each sequence f, so that the definition we have offered is equivalent to a (clumsier) variant satisfying the strict syntactic rules for a recursive definition (with auxiliary function a(m,f) = #f).

Having come this far, we can easily go on to define

Signed integers. It is convenient to encode the negative of an integer n as {{n}}, which leads to the following definitions:

$$-n ::= \text{if } n \in Z_+ \ \&\ n \neq 0 \text{ then } \{\{n\}\} \text{ else } \eta\eta n$$

$$|n| ::= \text{if } n \in Z_+ \text{ then } n \text{ else } -n$$

$$n *_1 m ::= \text{if } n \in Z_+ \leftrightarrow m \in Z_+ \text{ then } |m| * |n| \text{ else } -(|m| * |n|)$$

$$m -_{1.1} n ::= \text{if } n \in m \text{ then } m - n \text{ else } -(n - m)$$
(signed difference of positive integers)

$$m +_1 n ::= \text{if } m \in Z_+ \ \&\ n \in Z_+ \text{ then } m + n$$
$$\text{elseif } m \in Z_+ \ \&\ n \notin Z_+ \text{ then } m -_{1.1} (-n)$$
$$\text{elseif } n \in Z_+ \ \&\ m \notin Z_+ \text{ then } n -_{1.1} (-m)$$
$$\text{else } -((-m) + (-n)) \qquad \text{(signed sum)}$$

$$m -_1 n ::= m +_1 (-n) \qquad \text{(signed difference)}$$

$$m \geq n ::\leftrightarrow m - n \in Z_+ \qquad \text{(comparison)}$$

$$Z ::= Z_+ \cup \{-n : n : n \in Z_+\} \qquad \text{(signed integers)}$$

Since the operators $*_1$, $+_1$, $-_1$ extend *, +, and -

from Z_+ to Z, we can abbreviate them in the ordinary way as *, +, and -, relying on our knowledge of the types of their arguments to disambiguate any ambiguity which this might cause.

Rational numbers can now be defined in the ordinary way as sets of equivalent pairs of integers, i.e.:

$$ratc(n,m) ::= \{\langle x,y\rangle : x \in Z,\ y \in Z : x * m = y * n \ \&\ y \neq 0\}$$

$$Rats ::= \{ratc(n,m) : n,m : n \in Z \ \&\ m \in Z \ \&\ m \neq 0\}$$
(the rational numbers)

$$r *_2 s ::= \eta\{ratc(n_1n_2, m_1m_2) : n_1,n_2,m_1,m_2 : \langle n_1,n_2\rangle \in r \ \&\ \langle m_1,m_2\rangle \in s\}$$

$$r -_2 s ::= \eta\{ratc(n_1m_2 - n_2m_1,\ m_1m_2) : n_1,n_2,m_1,m_2 : \langle n_1,n_2\rangle \in r \ \&\ \langle m_1,m_2\rangle \in s\}$$

$$r +_2 s ::= r -_2 (ratc(0,1) -_2 s)$$ (rational sum)

$$r \geq_{2.1} 0 ::= (\exists m \in Z_+,\ n \in Z_+)\ n \neq 0 \ \&\ r = ratc(m,n)$$

$$r \geq_2 s ::= r - s \geq_{2.1} 0$$ (comparison of rationals)

Again, since $n \to ratc(n,1)$ is a natural 1-1 embedding of the signed integers into the rationals, we will feel free to abbreviate $ratc(n,1)$, $\geq_2$, $*_2$, $+_2$, and $-_2$ as n, $\geq$, *, +, -.

Having thus defined the rational numbers and the operations upon them, it is easy to go on to define

Real numbers and real arithmetic.

Real numbers are defined as Dedekind cuts:

$$Reals ::= \{x : x \subseteq Rats \ \&\ x \neq Rats \ \&\ x \neq 0 \ \&\ (\forall y \in x)(\exists z \in x)(z < y) \ \&\ (\forall y,z)(y \in x \ \&\ y \leq z \to z \in x)\}$$

$$u -_3 v ::= \{x - y : x \in u \ \&\ y \notin v\}$$ (real subtraction)

$$float(r) ::= \{x : x > r\}$$ (imbedding of rationals into reals)

$0.0 ::= float(0)$, $1.0 ::= float(1)$

$u +_3 v ::= u -_3 (0.0 -_3 v)$

$|u| ::= if\ u \subseteq 0.0\ then\ u\ else\ 0.0 -_3 u$

$u *_{3.1} v ::= \{x * y: x \in u \ \&\ y \in v\}$ (multiplication of positive reals)

$u *_3 v ::= if\ u \subseteq 0.0 \leftrightarrow v \subseteq 0.0\ then\ |u|_3 *_{3.1} |v|_3$

$else\ -|u|_3 *_{3.1} |v|_3$ (real product)

Note that the ordinary comparison of reals, $u \geq_3 v$, is simply $u \subseteq v$, and that the greatest lower bound glb(s) of a set of reals is simply Un(s). We write $u > v$ for $u \geq v \ \&\ u \neq v$.

$lub(s) ::= 0.0 - glb(\{0.0 -_3 x: x \in s\})$

$[a..b] ::= \{x: x \in Reals \ \&\ x \geq a \ \&\ b \geq x\}$

$(a..b) ::= [a..b] - \{a,b\}$

Again we choose to abbreviate $-_3$, $+_3$, $*_3$, $|u|_3$ as $-,+,*,|u|$, etc.

Continuous functions and integration.

Real_neighborhoods ::= $\{(x-\delta\ ..\ x+\delta): x \in Reals\ \&\ \delta \in Reals\ \&\ \delta > 0\}$

Opensubs(s,N) ::= $\{s \cap Un(t): t \subseteq N\}$

Real_contin(f) ::↔ $Df \subseteq Reals\ \&\ f \in Maps(Df,Reals)$

$\&\ (\forall s \in Real\_neighborhoods)$

$(f^{-1}[s] \in Opensubs(Df, Real\_neighborhoods))$

$f -_4 g ::= \{\langle x, f(x)-g(x)\rangle: x: x \in Reals\}$

$f *_4 g ::= \{\langle x, f(x)*g(x)\rangle: x: x \in Reals\}$

$const(r) ::= \{\langle x,r\rangle: x: x \in Reals\}$

$f +_4 g ::= f -_4 (const(0.0) -_4 g)$

$f \geq_4 g ::= (\forall x \in Reals)(f(x) \geq g(x))$

Step_contin(Reals) ::= {f: Df=Reals & (∃s,a)(#s∈Z_+ &

Real_contin(f|(Reals\s)) & Rf ⊆ [-a .. a] &Carrier(f)⊆[-a..a])}

Pos_lin_functional(J) ::↔

(∀f∈Step_contin(f),g∈Step_contin(f), c∈Reals)

(J(f)∈Reals & J(f-$_4$g) = J(f)-J(g) & J(const(c)*$_4$f) =c*J(f)

& f ≥$_4$ g → J(f) ≥ J(g))

Integral ::= η{J: Pos_lin_functional(J) &

(∀a∈Reals, b∈Reals)(b>a→

(J({<x, if x∈[a..b] then 1.0 else 0.0>: x:x∈Reals})=b-a))}

This last formula defines the ordinary integral, at least in a limited version. The theory of integration, and indeed much of analysis, is concerned with extensions and manipulations of this and related functionals.

To illustrate the level which has now been reached, we note that two of the most basic theorems of real analysis can be stated as follows:

Range Theorem: Real_Contin(f) & a < b →

f[[a..b]] = [glb f[a..b] .. lub f[a..b]] .

Heine-Borel Theorem: Un(s) =(Reals & (∀x ∈ s)(Opensubs(x,Reals)) &

a < b → (∃s_0 ⊆ s)(#s_0 ∈ Z & Un(s_0) ⊇ [a .. b])).

To complete this *excursus* into analysis, we will now say a few words about

Complex analysis. Here we define:

Cnos ::= Real × Real

u -$_5$ v::= <hd(u) - hd(v), tℓ(u) - tℓ(v)>

u +$_5$v ::= u -$_5$ (<0.0, 0.0> -$_5$ v)

u *$_5$v ::= <hd(u) * hd(v) - tℓ(u) * tℓ(v),

hd(u) * tℓ(v) + tℓ(v) * hd(u)>

$\bar{u}$ ::= <hd(u), -tℓ(u)>

```
|u|_5  ::=  η{x: x ∈ Reals & x * x = u *_5 ū}

Comp_neighborhoods ::= {{z: |z-u|_5 < δ}:
            u,δ: u ∈ Cnos & δ∈Reals & δ > 0}

Comp_open(s)  ::=  s ∈ Opensubs(Cnos, Comp_neighborhoods)

Comp_contin(f) ::= f ∈ Maps(Df,Cnos)
                  & (∀s∈Comp_neighborhoods)(f^-1[s]∈
                        Opensubs(Df,Comp_neighborhoods))

Cdiff(f)  ::= Comp_contin(f)
              & (∀z∈Df)(∃g)(Comp_contin(g) & Dg = Df)
              & (∀w∈Df)((f(w)-f(z))= (w-z) *_5 g(w))
                        (continuously differentiable function)

Analytic(f) ::= Cdiff(f) & Comp_open(Df)

f'(z) ::= η{g(z): g: Comp_contin(g) & Dg = Df &
              & (∀w∈Df)(f(w)-f(z) = (w-z) *_5 g(w))}

f'    ::=  {<z, f'(z)>: z ∈ Df}

C_integral(f) ::= <1.0,0.0> *_5 Integral(hd∘f) +
                              <0.0,1.0> *_5 Integral(tℓ∘f)

f *_6 g  ::=  {<z, f(z) *_5 g(z)>: z: z ∈ Df ∩ Dg}

Restr(f,a,b)  ::= (f|[a..b] ∪ const(<0.0,0.0>)|Reals\[a..b])

Line_integral(f,g,a,b)  ::=  C_integral(Restr(f *_6 g',a,b)) .
```

We are now in a position to state two of the main theorems of complex analysis, of which the first, Cauchy's integral theorem, requires a preliminary definition:

```
Nullhomotopy(f,h) ::=  Comp_contin(h) & Dh = [0..1] × [0..1]
  & (∃c)(∀x∈[0..1])(f(x)=h(<x,0>) & h(<x,1>) = c
      & h(<0,x>) = h(<1,x>)).
```

Cauchy's Integral Theorem: Analytic(f) & Cdiff(g)

& (∃h)(Nullhomotopy(g,h) & Rh ⊆ Df)

→ Line_integral(f,g,0,1) = <0.0,0.0>.

Open Image Theorem: Analytic(f) → Comp_open(Rf) .

The set-theoretic techniques whose classical uses have now received adequate review can be used with equal effectiveness to handle the combinatorial situations more typical of computer applications. To illustrate this fact, we shall simply define the basic notions of the theory of

Parsing:

Ordered_forest(t) ::↔(∀x∈Dt)(Seq(t(x))&(∀y∈(Dt-{x}))(Rt(x)*Rt(y)))=Ø

Steps(p) ::= {<p(i), p(i+1)>: i ∈ Un(Dp)}

Descs(t,x) ::= {p(#p-1): p: (Seq(p) & p(0) = x

& Steps(p) ⊆ {<u,v>:u,v:u∈Dt & v∈R(t(u))})}

Root(t) ::= η{r: Descs(t, r) ⊇ Dt}

Ordered_tree(t) ::↔ Ordered_forest(t) & (∃r)(Descs(t,r) ⊇ Dt).

The fringe of a finite ordered tree is the sequence of all its 'twig' nodes, taken in their left-to-right order. We can define this as follows:

f‖g ::= f ∪ {<n + #f,g(n)>:n: n ∈ Dg } (concatenation of sequences)

‖f ::= if ¬Seq(f) then Ø elseif #f=1 then f(0)

else (‖(f|(#f - 1)))‖f(#f - 1) (concatenation of sequence of sequences)

Fringe(t) = if t(Root(t)) = Ø then Root(t) else

‖{<n,Fringe(rd(n))>: n,rd: rd=t|Descs(t,t(Root(t))(n))

& n ∈ D(t(Root(t))) & #rd ∈ #t} .

We represent a context-free grammar, gram, simply as a set of pairs <r,ℓ> where r corresponds to the left-hand side of a production and ℓ, which must be a sequence, corresponds to the right-hand side of the production.

$$\text{Sentences(gram,rootsymbol)} ::= \{m \circ \text{Fringe}(t) : m,t:\text{Ordered\_tree}(t) \;\&\; \#t \in Z \;\&\; (\forall x \in Dt)(t(x) \neq \emptyset \rightarrow \langle m(x), m \circ (t(x))\rangle \in \text{gram} \;\&\; m(\text{Root}(t)) = \text{rootsymbol})\}$$

Beyond the definitional material presented in the preceding pages, the essential content of mathematics is a sequence of proofs of equality, predicate identity, and existence, which allow set-theoretic terms and predicates to be transformed from one form to another. These proofs rest upon algorithms, ideally as flexible and powerful as possible, which allow one formula to be deduced from a set of other formulae by sequences of single 'elementary' steps. Variants of these formal mechanisms reappear in the computer context as tools for algorithm verification and transformation, and will be discussed below.

B. From mathematics to algorithms.

To turn the mathematical dictions employed in the preceding section into computer programs, a succession of intermediate steps is necessary. Basically we must:

(a) eliminate all uses of infinite sets and

(b) eliminate all prohibitively expensive constructions, e.g. all but very cautious use of the power set operation, as well as all recursive definitions requiring inordinately many intermediate evaluations to converge.

Once these two steps have been applied to a mathematically defined function or predicate f, to yield a mathematically equivalent but generally less obvious and perhaps more complex form of f, we say that an *algorithm for f* is available. Concerning the essential work of accumulating and organizing such algorithms, we can make the following remarks:

(i) This work is in full swing. Even omitting numerical algorithms, well over a thousand significant algorithms have appeared in the literature, and dozens more are being published each month.

(ii) We will always be interested in algorithms which are as efficient as possible, and this will inspire us to derive

numerous equivalent algorithms, all calculating the same f. Some of these may calculate f only in special contexts, e.g. for arguments satisfying particular restrictions. Others may be specially adapted to efficient combination with other algorithms used in the same program, for which purpose the transformational technique reviewed below may be useful.

(iii) Functions f often appear in program loops,within which they are calculated repeatedly for argument values which change only 'slightly' between iterations. For this reason we are interested not only in algorithms which calculate f(s) *de novo*, but also in methods for calculating f(s') given f(s) together with some useful relationship between s' and s.

The large number of mathematical operations apt to be of interest for one or another application, the fact that variant algorithms for use in special contexts need to be studied, and the fact that we are interested in studying many useful combinations of algorithms can be expected to generate a very large algorithmic literature. As noted, this literature, already substantial, makes many useful mathematical operations available as primitives out of which applications can be composed. In its most abstract form, the armory of programming may then be taken to consist of the fundamental set-theoretic operations of mathematics, supplemented by the numerous mathematical operations for which algorithms are available. In the development of an application, these 'library' operations are expanded into algorithms that realize them, which are then further transformed and combined to attain higher efficiency.

The library operations used in this way will often depend on several variables having different sorts of values, and for efficiency's sake the algorithms which realize them will often iterate some correction or adjustment until the desired function value has been produced. Definition of these intermediate corrections may require use of various intermediate quantities. Thus the programmer will typically work with

a Cartesian product whose separate axes are spaces of different sorts, to which he will want to assign mnemonic variable names rather than numerical indices. Moreover, iteratively defined sequences, only the final component of which has any further use, will commonly be employed to express intermediate constructions. Hence arises the mechanism of variables, assignments, and while-loops that, together with the use of acceptably efficient set-theoretic and predicate expressions, characterizes the description of algorithms in our view.

Automatic treatment of every step of program development after the initial definition of a program's function is desirable. For this reason, techniques for automatic transformation of mathematical specifications into algorithms would be of considerable interest. In general, this problem is equivalent to and just as intractable as the problem of discovering mathematical proofs automatically. However, just as for mathematical proof, semiautomatic treatment of commonly occurring easy cases may be possible. Interesting heuristics for this, based on ideas which also find application in the automatic elaboration of simple proofs, have been suggested recently. Although these techniques are still highly experimental, we will outline a few of them.

(i) Formal integration. Suppose that S is a set and K(S) is a function of S which is to be evaluated; and suppose we know a mathematical identity of the form

$$K(S \cup T) = K'(K(S),T) ,$$

where K' is easier to evaluate than K, provided that T has only a few elements. Then the assignment x := K(S) can be transformed into the loop

```
x := K(φ);
(forall y ∈ S)
   x := K'(x,{y});
end forall;
```

which builds x incrementally. Similarly, a sequence of assignments $x_1 := K_1(S); \ldots; x_n := K_n(S)$ can be transformed into

```
x_1 := K_1(φ);  ...; x_n := K_n(φ);
(forall y ∈ S)
   x_1 := K'_1(x_1,{y}); ...; x_n := K'_n(x_n,{y});
end forall;
```

provided that we have $K_j(S \cup T) = K'_j(K(S),T)$ for $j = 1,\ldots,n$.

(ii) Use of standardized procedure templates. In some commonly occurring cases, we can force a logical predicate to have the value 'true' by applying a simple standard operation to the data objects appearing in the condition. For example:

$x \in s$ can be forced 'true' by executing 's := s ∪ {x}';

$x \leq y$ can be forced 'true' (if x and y are numbers) by executing 'if x > y then interchange (x,y)';

$(\forall x \in s)\ C(x)$ can be forced true by executing

```
(while ∃ x ∈ s | ¬C(x))
     force C(x) to the value 'true';
end while;
```

These rules, and others like them, can be compounded. Thus, for example, given a map f and a set s we form the transitive closure of f over s by forcing the condition $(\forall x \in s)\ f(x) \in s$ to be true. Applying two of the preceding rules in succession leads immediately to the standard elementary transitive closure algorithm

```
(while ∃ x ∈ s | f(x) ∉ s)
     s := s ∪ f(x);
end while;
```

Similarly, given a numeric-valued sequence, f, we sort it by forcing the condition $(\forall i \in (\#f - 1))\ f(i) \leq f(i+1)$ to be true. Again we can apply two of the preceding rules in succession to obtain the following standard 'bubble sort' algorithm:

```
(while ∃i ∈ (#f-1) | f(i) > f(i+1))
     interchange (f(i),f(i+1));
end while;
```

These observations reveal the existence of a class of 'fully trivial' algorithms derivable immediately by elementary transformation of the conditions which their outputs are to satisfy.

(iii) Incremental set constructions guided by minimality considerations. In some cases one will want to construct a set S satisfying a condition which can be written as $F(S) = \emptyset$, where F is a set-valued function for which a relationship $F(S \cup T) = F'(F(S),T)$ is available. In many such situations, it is useful, in attempting to reduce F(S) to the null set during an element-by-element construction of S, to attempt to minimize F(S) at each stage of the construction of S, always adding an x such that $F(S \cup \{x\})$ is not a superset of F(S). That is, we will build up S iteratively, always adding an x such that

$$(*) \qquad (\exists y \in F(S))\ (y \notin F(S \cup \{x\})) \ .$$

Sharir has recently shown that this kind of iterative construction of an S satisfying $F(S) = \emptyset$ is possible if F satisfies certain relatively mild monotonicity constraints, and that under related but more stringent constraints any sequence of x chosen to satisfy the condition (*) will lead via a loop of the form

```
S := ∅;
(while (∃x ∉ F(S)) (∃y ∈ F(S)) y ∉ F(S ∪ {x}))
     S := S ∪ {x};
end while;
```

to an S such that $F(S) = \emptyset$. A variety of interesting algorithms, not all of them entirely elementary, turn out to be derivable in this way.

C. Correctness proofs.

Whenever the mathematical definition D of an operation is replaced by an algorithm A which is supposed to realize the same operation, the question arises as to whether A and D

are in fact equivalent. Moreover, whenever program text initially containing separate algorithms $A_1,\ldots,A_n$ derived from separate mathematical operations $D_1,\ldots,D_n$ is transformed in a manner which combines these algorithms, the question of whether the transformed text will produce the same results as the initial program text must be faced. These are questions which the standard techniques for formal proof of program correctness are intended to answer. During application of these proof techniques, the basic definitions D_j appear as assertions in the algorithmic texts A_j, and one's aim is to formally prove that these assertions are valid whenever control reaches any point in A_j at which such an assertion is imbedded. Generally speaking, a notion of formal correctness always attaches naturally to any algorithm or program text which is used to implement a given mathematical operation or collection of operations at an enhanced level of efficiency.

No equally fundamental notion of correctness attaches to the externally motivated aspects of an applications program (which will be considered in more detail in the following section of this report). For example, we cannot expect to prove, in any formal sense, that the diagnostics issued by a parser are helpful or nonredundant, etc. (even though certain more limited statements about these diagnostics may be provable, e.g. the statement that no diagnostic will be issued at all if the text T being parsed is a valid sentence of an appropriate formal language; or even the more sophisticated statement that the number of diagnostics generated is no larger than the largest number of contiguous subsections of T which can belong to no valid sentence). However, once one has developed an initial very-high-level form of an externally motivated applications program and agreed that this program does deliver just the function that one wishes to specify, one will often proceed to transform this program to an equivalent but more efficient form. Hence the question of equivalence between several

program variants, which is a question amenable to formal approaches and to proof, arises even in situations initially dominated by informal, nonmathematical considerations.

These remarks suggest two lines of initial application for the formal techniques of program proof. First of all, one wants to take important operations which admit of very simple mathematical definition, but whose efficient implementation requires an intricate algorithm, and to prove the equivalence of algorithm and definition. In the relatively undeveloped present state of proof technology, attempts of this sort will be challenging enough to force numerous improvements in the techniques of program proof. A second significant direction suggested by the foregoing considerations is the development of *transformational* proof techniques which serve to relate differing versions of the same program. This is a valuable approach even for simplifying the proof of a single, mathematically flavored algorithm; and the above remarks suggest that it may be the sole method by which proof technology can be brought to bear on composite, externally motivated applications programs.

D. User-definable object types.

In natural language, objects are always classified implicitly into a variety of semantic 'sorts' or 'cases', and, as linguists have emphasized, this classification plays an important organizational role. Similar implicit classifications play an important role in mathematics, e.g. we can use the symbol '+' to designate both real addition and matrix addition, since we may know that x and y in x + y are reals, whereas A and B in A + B are matrices. Mechanisms of this kind have been much used in programming languages, where they appear as declarative systems for object 'typing' and (if the typing rules are rigid enough to ensure that the type of every object

is determinable during compilation) 'strong typing'. We emphasize that it is useful to provide such a mechanism of types directly at the mathematical level. Unless this is done, the only objects directly available in a very-high-level programming language will be sets, sequences, and objects such as integers, reals, etc. which have very straightforward mathematical definitions. Of course, objects of other kinds can be represented using these basic types. However, in developing a long program which is most naturally described as a sequence of manipulations of objects of a heuristic character not directly described by one of these fundamental types, it is best to push representational details into the background, and to think directly in terms of new kinds of objects and operations upon them. One will also want to extend the meaning of the language's handiest syntactic forms, i.e. prefix and infix operators, indexing f(x), indexed assignment f(x) := y, etc., and use these syntactic forms to designate operations on new types of objects. For example, in a program which makes heavy use of matrices, a + b and a - b should denote matrix sum and difference, but if a and b are bags this same syntax should denote bag sum and bag difference.

The *generic operator* mechanism used for this can be either static or dynamic. Dynamic mechanisms are more flexible and raise no union-type problems; static mechanisms have the advantage of allowing much useful type tracking independent of actual execution (execution may even be impossible).

Nevertheless, dynamically defined type have a mathematical significance which is as definite as that of statically defined types. From the dynamic point of view, a typed object is merely a pair x = <t,v>, where t is the 'type' of x and v is its 'value'. To apply any infix operator sign $\oplus$ to a pair of typed objects, one makes use of an auxiliary mapping mt, which maps every triple consisting of a pair of types

t_1, t_2 and operator sign $\oplus$ into a pair $<t_3,m>$ consisting of a result type and a map on a pair of subject values. Then an infix functional combination $x_1 \oplus x_2$ is interpreted as a synonym for the pair

$$<t_3, m(t\ell(x_1), t\ell(x_2))> ,$$

where $t_3 = hd(mt(hd(x_1), hd(x_2), \oplus))$ and $m = t\ell(mt(hd(x_1), hd(x_2), \oplus))$. A similar rule can be applied both to monadic operators written in prefix position and to special operators written in other syntactic forms, e.g. the syntactic form f(x) can be regarded simply as a special binary combination of f and x, having a value and type dependent on the value and type of f and x in the same way as any other infix operator.

Definitions allowing new meanings to be introduced for arbitrary infix, prefix, and special operator signs form a necessary component of such a system of user-definable object types. Such definitions can have the form:

(1a) for unary operators: *oplist typename by functionlist;*
example: * matrix by transpose;

(1b) for infix cperators: *typename oplist typename by functionlist;*
example: bag (+,-) bag by bagsum, bagdiff;

(1c) for k-parameter function applications:
typename(typename$_1$,...,typename$_k$) by functionname;
example: matrix (int, int) by matrixcomponent;

(1d) for k-parameter indexed assignments:
typename(typename$_1$,...,typename$_k$) :=
typename$_{k+1}$ by functionname;
example: matrix(int,int) := real by matrixassign;

In these definitions, the intended syntax of *oplist* and *functionlist* is as follows:

oplist → *operator_sign* | (*operator_sign* [, *operator_sign*])
functionlist → *functionname* | (*functionname* [, *functionname*])

Moreover, in a declaration (1a) or (1b), the *oplist* and *functionlist* are intended to have the same number of elements; and then the j-th element of the *oplist* involves the k-th element in the *functionlist*. E.g., in example (1b), x+y is handled as bagsum(x,y) and x-y as bagdiff(x,y), assuming that the values of x and y are bags.

New types can be introduced by declarations of the form

type *typelist*;

where the intended syntax of *typelist* is simply

typelist → *typename* |(*typename* [, *typename*]).

Typenames introduced in this way can be used as monadic prefix operators; the value of *type* <t,a> is simply the pair <*type*,a>. Of course, the 'initial' or 'base' language with which one begins will provide a built-in family of object types and of operators defined on these types.

To illustrate the use of these syntactic mechanisms, we shall consider their application to a small fragment of algebra, specifically, the introduction of polynomials as a new object type. This can be accomplished using the following declarations:

```
type polynomial;              /* introduces 'polynomial' as a type*/
polynomial (+,-,*,/) polynomial by
    poladd, polsub, poltimes, poldiv; /* introduces algebraic
                                 operations on polynomials */
```

It then only remains to define the few functions *poladd*, *polsub*, *poltimes*, and *poldiv*. Assuming that polynomials are represented internally as sequences of real coefficients, this can easily be done as follows:

```
poladd(a,b) ::= [if n ∉ D(set a) then 0.0 else (set a)(n)
                + if n ∉ D(set b) then 0.0 else (set b)(n):
                    n: n ∈ (D(set a) ∪ D(set b))];

polsub(a,b) ::= [if n ∉ D(set a) then 0.0 else (set a)(n)
                - if n ∉ D(set b) then 0.0 else (set b)(n):
                    n: n ∈ (D(set a) ∪ D(set b))];

poltimes(a,b)::= [ Σ {(set a)(j) + (set b)(k): j,k:
                    j+k = n & j ∈ D(set a) & k ∈ D(set b )}: n:
                      n ∈ (D(set a) + D(set b) - 1)];

degree(a)    ::= Un({n: n ∈ D(set a) & (set a)(n) ≠ 0.0});
                              /* degree of the polynomial  a */

leading(a)   ::= (set a)(degree a); /* leading coefficient of a */

procedure poldiv(a,b); /* polynomial division by repeated subtrac-
                                                          tion */
quotient := polynomial [0.0: n: n = 0];
                        /* initialize quotient to zero */
while degree a > degree b;
     monad :=[polynomial if n ∈ degree a - degree b then 0.0
               else leading a / leading b:
               n: n ∈ degree a - degree b + 1];
     a := a - monad b;
     quotient := quotient + monad;
end while;
return quotient;
end procedure poldiv;
```

Once these definitions have been given, polynomial computations can be carried out in their standard mathematical syntax which, of course, is our aim in this exercise.

3. External issues in program design.

A. Combining algorithms to create applications.

Algorithms (or more precisely mathematical functions for which algorithms are available) can be used to build applications, but are not themselves applications. Indeed, into the composition of any program which is to realize an application there will enter elements which reflect end-user aims whose content goes beyond the strictly algorithmic, and much, even most, of the design of a particular program may relate directly to this material. Design items of this sort typically represent the physical or administrative structure of real-world systems; the form and sequencing of expected input and desired output; the reactions, including prompts and warnings, expected from interactive systems; heuristic approaches held likely to manipulate material or symbolic objects in helpful ways; etc.

In programs, as they are ordinarily written, application-related material of this sort is inextricably intermingled with code fragments that realize the algorithms being used. However, sounder design practice would separate these two types of material more systematically, expressing design intent in terms of mathematical notions for which algorithms were known to be available, but at first suppressing all details concerning these algorithms. As an example of this, consider the problem of diagnosing syntax errors. For this, we can use the mathematical operations which, given an input string s, find the largest integer n such that s|n is the initial portion (resp. the middle portion) of a well-formed sentence. These operations, which given a grammar *gram* and root symbol *rootsymbol*, can be defined mathematically by:

longest_start(s) ::= Un({n: (∃t)(Seq(t) &
(s|n)‖t ∈Sentences(gram,rootsymbol))})

longest_contin(s) ::=Un{longest_start(t_1‖s)-Dt_1:t_1:Seq(t_1)}

and can also be realized by acceptably efficient algorithms. Closely related algorithms can be used to evaluate:

```
contin1(s) ::= {t(1): Seq(t) & longest_start(s‖t)
                      > longest_start(s)}
contin2(s) ::= Un({contin1(t‖s):t:Seq(t)}) .
```

In what follows it will also be convenient to use the sequence-truncation primitive defined by

$$s(n..) ::= \{\langle m, s(n+m)\rangle : m : m \in Z_+ \ \& \ n+m \in Ds\}.$$

Given these essentially mathematical primitives, we can design the intended diagnostic application as follows.

(i) It is assumed for simplicity that successive lines of input are read by an I/O primitive *readin* and appear as sequences.

(ii) We repeatedly find the largest part of the currently available input s which can be a portion (either start or middle, as appropriate) of a well-formed sequence.

(iii) If this is the whole of the remaining input, we have nothing to do. Otherwise, we print (an abbreviated variant of) the set contin1(s) (or contin2(s), if more appropriate), skip forward a few places in the input string in order to avoid redundant error messages, and then repeat from step (ii) as long as any input is available. This leads to the following definitions and code:

```
longest(s,erroryet) ::= if erroryet ≠ 0 then
                          longest_contin(s) else longest_start(s);
contin(s,erroryet)  ::= if erroryet ≠ 0 then
                          contin1(s) else contin2(s);
erroryet := 0;          $ initially no errors
input    := Ø;          $ initially null input sequence
(while (newline := readin) ≠ Ø) $ while input still exists
  print(newline);                 $'echo' line
  input := input‖newline;         $ add new line to existing
                                                      input
  (while longest(input,erroryet) < #input)
    print('***error***. one of following symbols required:')
    print(contin(s,erroryet));  $ print diagnostic
```

```
        input := input(longest(input,erroryet)+3,...);
        erroryet := erroryet + 1;        $ cumulate errors
        end while;
      end while;
```

We note that most of this code is externally rather than internally shaped. That is, all but a few details of this code are motivated by such nonmathematical, application-determined aims as the intent to generate helpful captions, print diagnostic messages in immediate proximity to the source string text point which generates them, etc. It is also to be noted that the body of text required to express this intent is comparable to that which suffices to define the integers or the real numbers and all basic operations upon them, and that to refine the elegance or psychological usefulness of the diagnostics we output, even to a limited degree, would require measurably more code. The sense in which design of the parsing application which we have just sketched goes beyond purely algorithmic issues should therefore be clear: the motivating aims (such as that of producing output which is helpful and thorough, but not too bulky) and assumptions (e.g., the assumption that the user will read his source text from left to right) that shape the code correspond to psychological rather than mathematical facts.

There has developed a large, though largely administrative literature concerning the important problem of how to come to terms with these important aspects of application design before the start of detailed programming. This is the so called problem of *requirements specification*. Concerning the literature devoted to this problem, the astute observer G. J. Myers comments: "The purpose of software requirements is to establish the needs of the user for a particular software product. Little can be said about the methods for verifying the correctness of requirements other than that the user is responsible for checking

the requirements for completeness and accuracy, and the developer is responsible for checking for feasibility and understandability. Although no methodology exists for external design, a valuable principle to follow is the idea of *conceptual integrity*, [i.e.]... the harmony (or lack of harmony) among the external interfaces of the system... The easiest way *not* to achieve conceptual harmony is to attempt to produce an external design with too many people. The magic number seems to be about two. Depending on the size of the project, one or two people should have the responsibility for the external design. ... Who, then, should these select responsible people be? ... The process of external design has little or nothing to do with programming; it is more directly concerned with understanding the user's environment, problems, and needs, and the psychology of man-machine communications. ... Because of its ... increasing importance in software development, external design requires some type of specialist. The specialist must understand all the fields mentioned above, and should also have a familiarity with all phases of software design and testing to understand the effects of external design on these phases. Candidates that come to mind are systems analysts, behavioral psychologists, operations-research specialists, industrial engineers, and possibly computer scientists (providing their education includes these areas, which is rarely the case)."

B. Software prototyping tools and application models.

It can be claimed that our inability, as noted by Myers, to specify requirements adequately is attributable in significant part to the lack of adequate software prototyping tools. It would surely be far better to create functioning, even if highly inefficient, system prototypes with which a potential user could experiment before work began on any much more efficient, and expensive, production system. Without this,

it is often impossible to say adequately in advance whether planned system features and responses will be found acceptable. The intended system user is, so to speak, in the position of a buyer forced to approve the design of a large commercial building by review of a voluminous written description of its rooms and fittings, without ever being able to see architect's renderings or a scale model.

Still worse, in the absence of software tools capable of handling the algorithmic side of an application in very abbreviated fashion, there is no way in which the would-be designer can push algorithmic concerns into the background in order to concentrate on the external design issues which may be central to effective treatment of an application. Although, e.g., the choice of features for an on-line editor, program maintenance aid, word processing system, industrial data-gathering application, graphics package, or natural language query system involves much art, there exists almost no literature aimed at examining or propagating the principles of this art, since at present its issues are hopelessly interwoven with the very different problem of describing the algorithms and low-level coding approaches in terms of which these applications will be realized. Only systematic use of a much higher level programming approach can remedy this deficiency and thereby replace the inchoate mass of 'know-how' and anecdote on which we now rely by organized, transmissible software engineering knowledge.

Such broad use of very-high-level programming and software prototyping tools could facilitate dissemination of the art of application design by examination of particularly successful examples. However, to bring the important, varied, and vexing problems of externally motivated program design more adequately under control, design techniques and tools of considerably more specific character are needed. One fundamental possibility is to develop special application-oriented programming languages whose objects and operations define useful standard

approaches to important application areas. A variety of such languages will be reviewed below. Two other suggestions can be advanced. The first of these, which also plays a role in the design of application-oriented programming languages, is to strive deliberately to use general mathematical operations rather than tailored special cases of them in developing prototype applications. This recommendation is illustrated by the 'diagnosing parser' code fragment given above, which uses very general parse-related mathematical functions (longest_start, longest_contin, etc.) to give a succinct and reasonably acceptable prescription for the generation of diagnostics. Contrasting with this recommended practice, ordinary application-oriented code tends to mix internally and externally motivated program material inextricably, i.e. output details are allowed to control the choice of algorithms, and opportunities to generate output which an algorithm seems to afford are allowed to determine much of what the end-user sees. The result is often an inartistic package which meets user requirements only minimally and which is full of redundant, hard to maintain, and inefficient algorithmic fragments. By separating external application design from choice and elaboration of internal algorithms much more cleanly, it should be possible to treat these two problems separately, and thus to arrive at more satisfactory solutions of both of them. (We note that the use of very-high-level programming tools can also contribute to this design goal.. Part of the reason for the unnecessary use of specially tailored algorithms in applications development is the difficulty of making separately developed nonnumerical procedures developed in languages of the Ada-PL/I-PASCAL level of language reusable. Indeed, structures which in a very-high-level language would appear as a handful of sets and maps turn at this level of language into mazes of subfields and pointers which it is hard to either learn about or use correctly. Programmers therefore tend to rework their routines rather than trying to interface to

library versions, and in reworking them the temptation to tailor them to whatever application is being developed often becomes overwhelming.)

A related suggestion is to use well-designed, relatively general-purpose application packages as building blocks in the construction of more complex applications. Consider, for example, the problem of designing an interactive system into which formatted commands will be entered to elicit system responses. As part of the design of such a system, command input conventions and command decomposition routines always need to be developed. It may be possible to handle this command input task by adapting a standard text editor very slightly. One possible convention is simply to take each user-supplied line prefixed by a blank as input to be appended to the end of a file being edited, but also to execute the line, as a command, unless it ends with an escape character. Lines not prefixed by blanks could then be regarded as editor commands, and the editor could also be supplied with a meta-command which executes a specified range of lines as a command. If this is done, the editor would also serve to define and implement command storage and modification facilities which would be as flexible and successful as the editor itself.

A related illustrative possibility is to handle command decomposition using a standard grammar-driven parse-and-diagnose routine which converts the command into a collection of abstract sets, sequences, and maps convenient for the next steps of processing. If this is done, the user-oriented details of recovery from and response to improperly formatted code can also be inherited from a preexisting package and need not be reinvented or reimplemented.

These examples illustrate the way in which well-designed, flexible application modules could be used, alongside of internally-oriented mathematical operations, as building blocks for more advanced applications. What is desirable here is an attempt to develop a library of application-oriented modules which

could be used in much the same way as a library of algorithms, but with the significant difference that the application-oriented modules would also embody pragmatic solutions to human-factors related problems.

C. Program transformation.

Once a sequence of mathematically defined operations has been assembled into a full application, there will intervene a sequence of transformations whose purpose is to improve the efficiency of the initial specification without changing any of its inputs or outputs. Automation of these transformations is highly desirable, since if they could be reliably automated, programmers could work entirely with succinct, very-high-level program designs, and could avoid involvement with detailed lower-level program forms whose maintenance tends to be very expensive. Unfortunately, our ability to automate the full range of transformations that needsto be applied for this goal to be reached is still rudimentary. This makes it likely that the production (though not the design) versions of programs will continue for a considerable period to require manual development through a spectrum of languages ranging from very abstract and mathematical languages at one end to languages of the PL/I-Ada level at the other. Nevertheless, work on automatic elaboration of higher level specifications can have some success and deserves to be pursued.

These transformational techniques have begun to attract considerable research interest, and by now quite a bit is understood concerning the most commonly occurring and useful transformations. Some of the most important transformational techniques are:

(a) formal differentiation of programs (Paige, Earley, Fong, Ullman, and Schwartz);
(b) replacement of recursion by iteration (Darlington, Burstall, Strong, Walker);
(c) replacement of data objects generated only to support iterations by coroutine-like 'generator'

procedures which produce pieces of these objects as required;

(d) replacement of nondeterministic choice operations, either by deterministic code sequences which make satisfactory particular choices, or by more restricted nondeterministic choice operations, known to be less likely to choose paths of exploration which will lead to subsequent failure (which would require backtracking) (Deak, Sharir); and

(e) replacement of some of the data items appearing in an initial program variant by code fragments which use other, unreplaced data items to recover the information which these data items would have carried.

Even though this short list includes many of the most commonly used high-level program transformations, it is unlikely that efficient algorithms which depend on these transformations can be produced automatically. To develop such programs, considerable user guidance will be necessary. Semi-automatic application of user-specified transformations, within the context defined by an interactive program manipulation system, is all that can be expected to become practical during the next few years. Even program transformation systems of this relatively limited capacity are substantially more complex than typical compilers, and will not be trivial to build. Moreover, it is likely that, until quite advanced transformational techniques are devised and successfully implemented, construction of programs by transformation will remain more expensive than direct manual program construction. Thus at first the only practical justification for a formal transformational approach is likely to lie in the fact that it can easily guarantee the logical equivalence of a series of program forms (so that, in particular, it can guarantee the correctness of a final form if the program form with which transformation begins is known to be correct). Nevertheless,

in spite of these *caveats*, it would be quite useful to develop mechanized program transformation systems, since their design can be expected to reveal many of the basic pressures which give programs their typical forms. We note also that the development of manually steered program transformation systems prepares for future attempts to design more fully automatic systems.

As an illustration of the nature of the transformational techniques that have recently begun to be developed, we will consider an important and much-studied type of algorithm, namely a compacting garbage collector. From the abstract point of view, this algorithm takes as input a map P representing a storage layout. The domain of P is assumed to be a set of integers I representing addresses, and P maps each of these integers I into a finite sequence of integers, representing the contents of the storage block which begins at I. We make the following three additional assumptions:

(i) each of the sequences S (storage blocks) constituting the range of P is of nonzero length;

(ii) each integer in such an S belongs to the domain of P (i.e., 'is the index of some other storage block'); and

(iii) each integer I in the domain of P is the sum of the lengths of all preceding storage blocks (i.e., in building up the storage layout P, storage has been allocated sequentially).

In describing the garbage collection algorithm it will be convenient to make use of a few additional primitive operations and notations. The necessary primitives have the definitions:

$\cap s ::= \{x: x \in Un(s) \ \& \ (\forall y \in s)(x \in y)\}$ (intersection set)

$\Sigma f ::= \text{if } Df = \emptyset \vee Df \not\subseteq Z_+ \text{ then } 0 \text{ else } f(\eta Df) + \Sigma\,(f|(Df - \eta Df))$

(sum of the values in the range of f).

Since map definitions of the form

(*) $\{\langle x,e\rangle : x : C\}$,

where e is any term and C any predicate expression, occur frequently, it is useful to allow them to be written in the convenient abbreviated form

$[e : x : C]$.

Garbage collection and compaction consist of the following steps:

STEP 1. Compute the set U of all addresses of active cells in the domain of the storage map P, i.e. the set of addresses of cells reachable by a sequence of 0 or more pointers starting from an address in a given 'root set' (which for simplicity is assumed to contain the single address 0; the reason for this assumption is that if it contained other addresses, it might be possible for the corresponding cells to have moved after compaction so that the new addresses of all root nodes would also have to be returned from our garbage collector. By forcing the root set to contain the single address 0 we guarantee that after compaction, the root will still be located at 0).

STEP 2. Calculate a map N which maps each (old address of an) active cell (i.e. each cell whose address is in U) to its new address. The new address of an active node b is the sum of the lengths of all active blocks whose address is smaller than that of b.

STEP 3. Calculate the new storage layout and assign it to P. The new layout consists of a succession of blocks which appear at their new addresses and contain integers that point to the same blocks as before, but at their new addresses.

These operations can be programmed in just three lines:

```
U := ∩ {x: x∈pow(D(P)) & 0∈x & (∀a∈x, c∈P(a))(c∈x)};   $step 1
N := [ Σ [#P(c):c:c∈U&c<b]:b:b∈U];                    $step 2
P := {<N(a), N∘(P(a))>:a:a∈U};                        $step 3.
```

Having this short program in hand, we can apply a sequence of correctness-preserving transformations to it: Step 1 can be expanded into a standard transitive closure algorithm which builds U incrementally, starting with {0}, and adding addresses to U as long as there exist addresses in U whose blocks contain pointers to cells whose addresses are not yet in U. It is easy to show that this procedure yields the desired U. Step 2 can be expanded into a loop in preparation for a loop-fusion transformation.

Step 3 can be split into 2 substeps; in the first of these substeps we can adjust the pointers in the blocks to point to their new addresses without moving the blocks themselves yet. Then the second substep must move the adjusted blocks to their new addresses. This splitting prepares for the fusion of Step 3 with Step 2.

This leads to the following somewhat lengthier variant of the garbage collector algorithm:

```
U := ∅;
(while (∃b∈U, a∈P(b)) a ∉ U)                    $ step 1
     U := U ∪ {a};
end while;
N := ∅;
(forall a ∈ U)                                  $ step 2
     N(a) := Σ[#P(c):c:c∈U & c<a];
end forall;
Q := [N∘(P(a)):a:a∈U];                          $ step 3.1
P := {<N(a),Q(a)>:a:a∈U};                       $ step 3.2
```

Next the while loop in Step 1 can be changed so that instead of testing if there exists an element having a particular property, it defines the set W of all elements having that property and tests if $W \neq \emptyset$. This allows us to apply formal differentiation, i.e. to maintain the value of the set W and to update this value incrementally whenever any of the parameters on which it depends are changed.

Step 3.1 can be fused into the loop that calculates N (i.e. into Step 2). This transformation is justified because the loop computing N does not depend on Q and the computation of Q does not have any side effects. Once this is done, the value of Q can be produced incrementally within the loop into which the calculation of Q has been placed. The single statement of Step 3.2 can also be expanded into a loop.

These rather more elaborate transformations bring us to the following third version of the garbage collector algorithm.

```
U := Ø;  W := {0};
(while W ≠ Ø)                                          $ step 1
     a := ηW;  W := W \{a};  U := U ∪ {a};
     W := W ∪ {c:c∈P(a)&c∉U};
end while;
N := Ø;
Q := [Ø:b:b∈U];
R := {<P(c)(i),<c,i>>:c,i: c∈U, i∈#P(c)};
(forall a ∈ U)                  $ R is an auxiliary map which sends each
     N(a):= Σ⌈#P(c):c∈U&        $ 'address' into all of its occurrences
                 c < a⌉;        $ step 2, 3.1, and definition of
     (forall <b,i>∈R{a})        $ the 'memo map' R
        Q(b)(i):=N(a);
     end forall;
end forall;
P := Ø;                         $ step 3.2
(forall a ∈ U)
     P(N(a)) := Q(a);
end forall;
```

These transformations can be continued, and would lead after just a few more transformational cycles to a fully fleshed-out and quite efficient garbage collector code.

It is also interesting to note that many of the garbage collector procedures that have appeared in the literature can be derived from our three-line initial version simply by varying the transformations applied to it. Thus the transformational approach to program development is both a tool for algorithm discovery and a way of improving our understanding of program structure by exposing the common root of codes that at first sight may seem quite unrelated.

D. Application-related language features.

We have noted that most of the code put together to implement a given application will relate to external requirements of the application rather than to internal algorithmic issues. When this is the case, and once some stable logical pattern has been found in at least some of the more important operations typifying an application area, it becomes appropriate to develop a special application-oriented language for the area. At its best, such a language will define powerful conceptual tools for attacking the characteristic problems of the area, so that even unimplemented languages of this kind can be useful instruments of thought.

Moreover, definition of an adequate semantic framework for an area makes it possible to compare broad abstract (but still precise and formal) programming problem solutions with typical hand-optimized solutions. Once this has been done, development of analysis/optimization techniques which aim to bridge the gap between these two styles of solution can begin.

In approaching any important specialized application area, it is the task of the language (or 'semantic mechanism') designer to find a harmonious family of formally well-defined operations and objects in terms of which its most characteristic problems can be handled conveniently. These should include methods for receiving and analyzing inputs in the form typical for the intended application area; for performing

required manipulations and calculations; and for reacting to and protecting against errors, managing states of partial information, and generating responses at appropriate times and in appropriate formats.

In well-established application areas in the physical sciences and engineering, standard mathematical tools and notions for handling major problems will often be available, and then the task of programming language design may simplify considerably. In less classical areas, discovery of the right notions around which to organize one's attack on an application area can be quite challenging, and structures having no precise analog in classical mathematics will often be appropriate.

Various successful application-oriented languages illustrate these points.

(a) Work on languages like Concurrent PASCAL, MODULA, and Ada has begun to define an adequate semantic framework for concurrent process and real-time programming, an area of very special interest in the design of operating and device-control systems. Software of this type must manage numerous sensors, effectors, storage devices, and analysis routines, all running in parallel and subject to real-time constraints. In the absence of an organizing conceptual framework embodied in appropriate special-purpose languages, such software has been notoriously difficult to put together. The leading ideas of these languages, whose importance is now generally recognized, are reviewed in a separate section below.

(b) A few special-purpose languages (e.g. GPSS, SIMULA, SIMPL) have supported simulation-oriented pseudoparallelism. This semantic mechanism provides a restricted parallel-process environment, differing significantly from that required to describe fully parallel environments of the operating system type, but adequate to state the causal rules of a simulated

universe whose elements interact via events that take place at successive, discrete intervals of time. Once a model has been defined by giving a set of rules in such a language, the model can be run, can generate outputs, and statistics concerning its internal activity can be collected.

Though still largely undeveloped for this purpose, languages of this type might be ideal vehicles for prototyping commercial applications, which in effect create computerized models of the activities of large firms or other relatively decentralized organizations that react to externally generated data stimuli and function by transmitting messages internally. In a language supporting pseudoparallel processes, such applications can be created by writing a collection of short independent programs, each of which defines the action of one particular kind of 'clerk' or 'processing station', that is, the state or file changes triggered by incoming documents or signals, and the contents and further destinations within the system of messages generated at such 'processing stations'. Program fragments of this kind could be quite close to the definitions of operating procedures which a firm is accustomed to use, so that it ought to be considerably easier to write programmed descriptions of a firm's procedures using pseudoparallelism than to create a business application of standard form, in effect by serializing a description which is most naturally parallel.

(c) Continuous system simulation languages, such as DYNAMO, accept ordinary differential equations as input and are able to set up solution algorithms for these equations and make the resulting solutions available either in graphic form or as input to procedures performing further analysis. Various languages and program packages used for circuit analysis are related to these continuous simulation languages, but carry them one step further by making it unnecessary to set up the differential equations which describe a circuit manually. Instead, one simply writes a formal description of the circuit as a

collection of resistances, compactances, inductances, and active elements of known characteristics. From these, the differential equations which describe the circuit are generated automatically.

This same notion, that of a language specialized to facilitate the description of engineering objects, whose characteristic equations are then generated automatically, leading to automatic calculation of object parameters and reactions, is also fundamental to such mechanical engineering languages as those described below:

(a) Graphics, animation, and machine-tool control languages illustrate another major application-oriented theme: direct description and manipulation of two- and three-dimensional geometric objects. A full-scale animation language will also allow direct manipulation of paths and rates of motion, viewing angles, levels of illumination, and surface color and reflectance. Machine tool control languages like the widely used APT language also facilitate the description of curves and surfaces, but also include mechanisms for automatic translation of these descriptions into cutting-tool paths of motion which will "sculpt" these surfaces in metal. Related semantic mechanisms play a role in various experimental robot-manipulator control languages and software packages.

(b) Various other application-oriented languages, e.g. string-and-pattern-oriented languages like SNOBOL; languages for computer controlled typesetting, including the description of complex mathematical formulae; design automation languages; lesson writing languages; etc., could be cited. The common attempt of these languages is, as we have said, to define a semantic framework encompassing the most typical objects and operations of the application area to allow the characteristic dictions of the application area to be used directly as programming language statements; and to avoid any demand for extensive or expanded detail where practitioners of the application make use of implicit understandings or succinct, declaration-like statements. To reach these goals, familiarity with the intended application will be required, since different applications may require a wide variety of different syntactic and semantic approaches.

E. Operating system languages.

Languages, like Concurrent PASCAL and MODULA, which are intended primarily for the description of operating and real-time device-control systems, have been actively investigated and developed during the last five years. These languages seem destined to play principal roles in that standardization of software interfaces on which easy, flexible attachment of computational nodes to future networks will depend. For this reason, we will review the basic features of these languages, and the underlying semantic requirements which shape these features, at some length in the present section.

An operating system (or 'parallel process') language must be concerned with a much broader range of issues than a 'monoprocess' language of the conventional type. The principal issues to be faced are roughly as follows:

(a) Control of multiple processes. Operating and real-time systems are ultimately used to control external devices which have their own inherent delays and timing constraints. For this, an environment of cooperating parallel processes, which can be alternately executed, suspended, and resumed, is appropriate. These processes will communicate through shared data objects. For this to be possible, one must ensure that each process using a shared data object O leaves O in a consistent internal state when its manipulations of O are complete, so that processes needing to use O subsequently can be sure that they will find it in a consistent state when they begin to access it. It is therefore appropriate to associate the code which manipulates O with the object O itself. This approach, pioneered in the SIMULA language, and subsequently advocated by Hoare, defines the kind of object to which Hoare has given the name 'monitor', namely a package of data structures and

of routines which have exclusive rights to manipulate these objects. Once a process has invoked one of these routines, it gains exclusive rights to modification of the object and prevents any other process from reading the object; however, multiple read-only accesses to the object can be in progress at any one time. This fundamental semantic requirement can be administered in several ways, e.g. by:

(a) locking the object as soon as an access routine is invoked, or

(b) recording all object modifications in a process-local data area until return from the object-associated access routine ('end of transaction') at which time all updates are finalized and all other processes currently accessing the object are pushed back to the start of their access.

Another significant characteristic of operating systems which operating system languages must reflect follows from the fact that not all the processes which such a system must accommodate will cooperate successfully with each other. Since some of these processes will correspond to user programs in a state of development, which may be incorrect or even deliberately malicious, processes written by different authors will be 'mutually suspicious', and will therefore prefer to interact only via rigorously defined interfaces whose conventions are rigidly enforced. A well-developed operating system language must therefore provide mechanisms which determine what system elements each process is allowed to access and the manner in which access privileges originate, can be shared, are made available temporarily or permanently to other processes, etc. Moreover, an operating system, as distinct from an ordinary program, must be able to compile and execute indefinitely many new programs as they are defined by system users. Operating system languages therefore need to support dynamic compilation and must include rules that relate names which occur in newly compiled codes to those which occur in code compiled earlier.

Various other semantic issues with which operating system languages must deal deserve emphasis:

Resource allocation and recovery. An operating system needs to ensure that no process gains control of so large a portion of system resource as to impede continued system functioning or degrade efficiency. Resources made available to user-level processes must be fully recoverable, either when the process terminates normally or when malfunction is detected, in which case smooth procedures for preemptory eviction of malfunctioning user processes must be available. These resource-recovery and eviction mechanisms must be foolproof and quick-acting.

Process urgency, real-time control, and preemption. The processes which an operating system manages will always contend for the limited execution resource which the system makes available. Thus the system will have to choose a few 'most urgent' processes for immediate execution, leaving other processes to wait. However, process urgencies can shift drastically in response to external signals. For example, a disk read process which cannot move forward at all during the lengthy period in which a disk is rotating to a readable position will suddenly become urgent once an inter-sector gap arrives under a read/write head. An operating system language must therefore provide "preemptive message" or "interrupt" primitives which allow the effective priorities of process to respond rapidly to external signals. If high-speed response to external events is to be possible, the language implementation must avoid expensive internal linkages, and cannot permit situations to arise in which urgent processes are blocked by less urgent processes' occupancy of needed data objects. To improve the efficiency of the most urgent and/or frequently executed portions of a concurrent process system most of which is to be written in an operating system language, two approaches are possible.

A familiar technique (unfortunately, the only one which is presently feasible, given the relatively undeveloped state of operating system languages) is to allow urgent system portions to be written in a lower level language (e.g. assembly language). Portions for which this has been done must then interface to the remainder of the system at the implementation level of the operating system language, and must appear to the remainder of the system as built-in routines which conform to the conventions of the operating system language in all detectable ways. It is clear that to do this a programmer needs to know all relevant internal implementation details of the operating system language. A more satisfactory approach, but one beyond the present state of the art, would be to apply global analyses like those used in optimizing compilers to determine which of the internal synchronizations and security checks of the ordinary language implementation can be omitted for a given, relatively isolated collection of processes and data objects. For this to be possible, it is necessary to submit the whole set of processes and objects comprising a systems 'urgent core' to an analyzer which could extend its analysis not only intraprocedurally but also between processes. Since the processes allowable within such a core must all necessarily be relatively simple, it is reasonable to expect that such analysis can be penetrating enough to eliminate much wasted motion without becoming unduly expensive.

Recovery, reliability, and concurrent use of files. If an operating system is successful, the files which it stores will quickly grow to be more valuable than the machine on which the system executes. This implies that the integrity of those files must be guaranteed even after physical failure of the computing hardware, and even after loss of fairly extensive parts of the storage system, occasioned, e.g., by disk head crashes. Moreover, since these files will contain valuable and often unique information, many applications will contend for their use. Accordingly, the file design ideally should allow file query to proceed in parallel with

file update, while preserving logical consistency, and should somehow ensure that no data state is ever reached which will make recovery impossible if physical failure suddenly occurs. Finally, concurrent file use must never lead to irresolvable deadlock.

The preceding list of operating system issues makes it quite clear that operating system languages deal with a challenging complex of semantic issues. These issues generally require centralized solution at the language or system level, since most of them have to do with inter-user contention or suspicion, and hence with problems that no individual system user either wants to be involved with or can manage by himself. The recently developed operating system languages have begun to resolve some aspects of this thicket of problems. One main contribution to date is Hoare's monitor notion, which provides an attractive basic framework for coordinating access by multiple processes to shared data objects. (More recently the Ada language design group has proposed a related mechanism, the 'rendezvous', which allows more flexible programmed control over the sequence in which contending processes are allowed to use a shared data item.)

We emphasize however that this work serves only to cast an initial light on a collection of problems which will require sustained work for decisive solutions to emerge. The Ada mechanisms, like the simpler Hoare monitor notion which they generalize, only provide basic synchronization tools, together with a simple, priority-driven scheduling regime. However, they do not address any of the other problems which we have reviewed, i.e. isolation of mutually suspicious processes, resource control, dynamic recompilation and extension of portions of a running system, reliability in the presence of hardware failures, or the problem of providing flexible, high-concurrency access to large data objects (like files), of which several may have to be accessed in a coordinated way by one or more processes, and in a manner guaranteed to preserve important global invariants.

3. Pragmatic issues.

We conclude our survey by reviewing various pragmatic language features which implementers of future programming languages should be encouraged to provide.

A. Aids for measurement of program behavior.

A program's expenditure of time is normally irregularly distributed, e.g. very little time may be spent in long, complex sections of code which have received much programmer attention, while inconspicuous loops performing relatively trivial character-move or data-buffering operations can consume substantial fractions of total execution time. For this reason, a programmer will normally find it difficult to determine accurately what parts of his program are efficiency critical. Without this information, he can easily complicate a program unnecessarily in a misguided attempt to make it more efficient which actually gains only marginal advantages while overlooking simple changes that can have much greater efficiency impact. To avoid these pitfalls, and allow a clear focus on those transformational directions or changes of language level likely to have real impact, well-designed measurement utilities, built directly into a language's compiler and run-time support system, are required. These utilities should produce information on run-time program behavior which is then related back to a program's initial source listing, where it should appear as a set of markings and numbers which are easy to scan and absorb visually. Information of this sort can also be useful in the early stages of debugging.

The precise nature of the information generated by a run-time program measurement system will depend on the level of the language in which programs to be measured are written. For a relatively low-level language, the following items will normally be relevant:

(a) fraction of time spent in executing each statement,
(b) division of time between internal and I/O operations,
(c) number of iterations through each loop, and
(d) success/failure ratios for each branch point.

If the program runs in a paging environment, one will also want to measure:

(e) percent of page faults occurring at each program location, and
(f) program or data location accessed by the operation which causes each page fault.

For very-high-level languages, one will want to measure a broader range of quantities. Programs in such languages will ordinarily compile into code one part of which is directly executed while another part consists of calls to support library routines. Moreover, such languages will generally execute in a garbage-collected data environment, making it necessary to relate the general overhead of garbage collection back to the program points at which data objects are being created. Accordingly, for very-high-level languages, one will want to measure:

(a) the actual distribution of execution time over the program, allowing for the time actually required to execute each particular instruction (this time can be highly variable);

(b) the number of calls to offline library routines (This information can be important to a user trying to assess the adequacy of a data structure design.); and

(c) the places at which space is being allocated, possibly needlessly. Time required for garbage collection deserves to be charged against each instruction in proportion to original allocations of space. Moreover, points of excessive space allocation can pinpoint failures in copy-elimination or data-conversion mechanisms.

All the sorts of information alluded to above can be collected in a fairly uniform way by attaching small groups of counters to each of the basic blocks of a compiled program, and by incrementing these counters appropriately. These

counters, and the rules for incrementing them, completely define a system of measurements. At the end of execution, the counters should be examined and printed out (perhaps in a bar chart representation) in appropriate relationship to the source text of the program which generates them.

The following counts and incrementation rules correspond to the measurements described above.

(i) Block entrance count. Incremented each time a block of very-high-level source code is entered.

(ii) Block-calls-library count. Each time such a block is entered, we can set a global *present block indicator* variable PBI to a corresponding value. Then, each time the library of run-time support routines is called, we increment BCL(PBI) by 1. The resulting data profile will show the extent to which sections of very-high-level code need to call the underlying support library.

(iii) Space allocation profile count. Each time the space allocator is called to allocate N words of heap space, we execute SAPC(BPI) = SAPC(PBI) + N. The resulting data profile will show the extent to which high-level code sections force the allocation of storage.

(iv) Execution time-consumption profile. When block PBI is entered, execute ETP(PBI) = ETP(PBI) + K, where K is the number of instructions comprising the block. When the run-time support library is entered, we can count the number of instructions executed. On each return from the support library, we can increment ETP(PBI) by the total number of instructions executed since the library was called. This will generate a profile, related to profile (ii) above, but showing the complexity, rather than simply the number, of support library calls.

B. Debugging

Debugging always starts with evidence that a program error has occurred somewhere in the history of a run. The problem in debugging is to work one's way back from the visible symptom to this program error. What one seeks can be called the *error sources* or *primal anomalies*, which are those wrongly stated operations or tests whose immediate consequence is the transformation of a collection of reasonable inputs into an output which is unreasonable in some regard. Of course, the history of an extensive computation constitutes a vast mass of data, impossible to survey comprehensively. The debugging process therefore aims at the exploration of as narrow a path as possible, with the aim of finding one's way back to one or more primal anomalies.

Here it is interesting to compare the two quite different processes of syntactic and semantic debugging. Even if we assume that raw program text (carefully desk-checked but never compiled) may contain as many as 1/10 syntax error per line on the average, the syntactic debugging of a 1000-line program normally proceeds routinely and rapidly. The tool that allows this is a compiler with fairly good syntactic debugging aids, among which the following are particularly desirable:

(a) unambiguous, easy-to-comprehend error messages;
(b) suppression of spurious error messages generated by prior errors; and
(c) a diagnostic capability which does not decay during the parsing of a lengthy, error-rich text.

These capabilities lie well within the present state of the art of parsing. If a compiler with these capabilities is available, the normal syntactic history of a 1000-line text initially containing 100 errors would ordinarily be something like the following:

Compilation 1: 125 error messages generated, of which 75 are genuine; 75 errors corrected, of which 10 are corrected wrongly.

Compilation 2: 70 error messages of which 30 are genuine; 30 errors corrected, of which 5 are wrongly corrected.

Compilation 3: 20 error messages of which 7 are valid; 7 errors corrected, of which 1 is corrected wrongly.

Compilation 4: 10 error messages, of which 4 are genuine. All 4 errors successfully corrected.

Compilation 5: No errors.

In an interactive system providing rapid turn-around, this need not take more than a few hours. Note the important role played by the ability to uncover multiple faults during a single run.

Next consider the process of semantic (i.e.'logical' or 'execution') debugging of the same program. Here we make the more favorable assumption that, owing to careful desk-checking and to the elimination of some logical errors during syntax checking, only 50 errors are present in the original 1000-line program. Now the typical iteration is approximately as follows.

(a) The program runs and bombs. Assuming that a miscellany of print statements was included for debugging purposes, the programmer then forms an idea of what has happened (e.g. certain code never reached, wrong argument values passed to certain procedures, unreasonable values detected for certain variables).

(b) This evidence, analyzed, will in favorable cases point the finger of suspicion at certain narrow program sections. However, in unfavorable cases, the available evidence may be

quite ambiguous, and may simply lead the programmer to generate considerably more extensive traces and dumps. Three typical cases can be noted.

(b.i) Within a region of code described as suspicious, at least one visibly incorrect instruction might be spotted and corrected.

(b.ii) A program region containing the error may be correctly described, but no specific error located. In this case, one more run, with denser tracing in the error region, can locate the anomaly.

(b.iii) The program region first suspected may in fact contain no error. In this case denser tracing will simply confirm the good behavior of the suspected region, after which reconsideration may lead to suspicion being cast, this time more correctly, on some other region.

Accordingly, the following are reasonably typical sequences of steps in uncovering a logical error.

Step 1: Suspect region R, insert traces.
Step 2: Locate and fix bug.
Step 3: Correct syntax error in step 2. (Bug is now fixed).

Alternatively:

Step 1: Suspect region R, insert traces.
Step 2: Region R ok, suspect region R', insert new traces.
Step 3: Correct syntax error in step 2, obtain new traces.
Step 4: Locate and fix bug from new traces.

Overall it can be hard to fix more than 1/3 bugs per run, as compared to the estimated average of 25 bugs fixed per run in our hypothetical account of syntactic debugging. Thus 150 runs, which might represent as many as 10 days work, can be required to fix the 50 logical bugs which might very typically be present in a new, 1000-line program.

To alleviate this vexing but all-too familiar situation, we must aim to transfer as much of the debugging as possible from the execution phase to the more productive compilation phase, increase the probability of finding at least one logic bug per run (if any is present), and make it possible to find more than one bug per run. The following considerations are directed toward this end.

(i) Global analyses capable of detecting program anomalies such as uninitialized variables, unused computations and type errors should be applied routinely during compilation, and the results of these analyses should be used to generate diagnostics.

(ii) It is well worth increasing the redundancy of program text in ways likely to expose errors during compilation. A primary technique here is declaration of variable types and of the types of input and object arguments which each operator will expect and produce. (If this information is used only for error detection, and not to steer compilation or execution in any other way, then inherently ambiguous cases can be handled simply by declaring particular variables to have ambiguous 'union' types. This will sidestep the ambiguity, with some small loss of diagnostic precision, but without creating any further complications.) Every subroutine and operation should then indicate the type of inputs which it expects and the type of output which it produces.

(iii) It is best that programs should not run for long after they have begun to generate erroneous quantities, since the longer they run the more remote the primal anomalies will become. Two techniques can be used for this:

(iiia) Data items should be dynamically type-tagged, and each type-error should lead to the generation of diagnostic information.

(iiib) A program being debugged should be thickly larded with dynamically checked assertions. If this is well done, the probability that a logical error will lead to quick blowup should be quite large.

(iv) Enough information to make it possible to trace back to a primal anomaly should be dumped routinely upon program blowup. What seems desirable is to dump the *last* value assigned to each variable X by every statement that modifies X. If this is done, a primal anomaly will only be hidden if the instructions I which constitute it generate an erroneous result R, pass R along as inputs to the instructions which will eventually (and probably soon) develop an error symptom from R, following which I is some how reexecuted, this time producing a correct-looking result which hides the erroneous character of I. Such tricky situations are of course possible, but unlikely.

To generate such a comprehensive dump of last values assigned, we can proceed as follows. As a program P runs, a count can be kept of the number of times each basic block within it is executed. If and when P fails, these counts will be available. The program can then be executed again in 'debug mode' and these counts decremented as execution proceeds. Each time a count reaches zero we know that we are entering a block for the last time. Wherever this happens, we can switch the block into an alternate mode in which variables are printed each time they are modified (along with an indication of the statement which is affecting the modification). The cost of this is only a doubling of the normal execution time of an erroneous run, which is probably a smaller cost than would be incurred by the less systematic process of ordinary debugging.

On failure it is also appropriate to dump an indication of routines currently invoked, with the values of their parameters, and of control-flow history. This history can consist of a statement of all branches recently taken, with an indication of the number of times taken if a given branch is taken repeatedly in the same way.

Next we turn to the question of how to discover more than one bug per run. The simplest technique is to generate diagnostic information whenever an error (e.g.

a dynamic type fault or a violated assertion) is detected, but to let a run proceed until some error limit is exceeded. Especially in the early phases of debugging, errors will tend to be independent, so that this approach will generally reveal multiple independent faults.

Another more sophisticated approach to discovery of more than one anomaly per test run is worth suggesting. Ordinarily, quite a few features of programs are generated by an implicit optimizing process of 'set-theoretic strength reduction' or 'formal differentiation' discussed earlier in this report. This optimization introduces variables x which carry the values of expressions $e(y_1,\ldots,y_n)$ that would otherwise have to be calculated repeatedly, but makes it necessary to update the value of x whenever $y_1,\ldots,y_n$ are modified, a requirement that can easily lead to error either because an update operation is forgotten or because it is wrongly expressed. In this situation, there will naturally arise assertions of the form

$$\text{ASSERT: } x = \text{expn } (y_1,\ldots,y_n) \ .$$

We can then change the syntax of such assertions to

$$\text{ASSERT: } x := \text{expn } (y_1,\ldots,y_n) \ ,$$

and agree that assertions having this latter form which fail will generate appropriate dumps but assign *expn* to x and continue execution. In many cases, this will allow defective programs to continue correct execution, up to the point at which one or more additional anomalies are uncovered. To generate the necessary dumps without increasing execution costs significantly, an execution count technique generalizing that outlined above can be used.

c. Linking software systems.

The growing size of fast memories and the availability of new large virtual memories suggest that it may soon be feasible to develop ambitious software systems which incorporate many separately developed programming languages and applications packages and combine their facilities. Thus, for example, we can imagine a combined SETL/SNOBOL/LISP/APL MACSYMA/EISPAK/ ... system for combined symbolic and efficient numerical processing. In the following paragraph we will sketch a technique for organizing the interfaces necessary for the development of such large hybrid systems. Note that the systems mentioned here not only incorporate a wealth of carefully designed algorithms but also embody important human factors designs which can aid the programmer greatly in dealing with particular mathematical and applications areas.

We observe, to begin with, that only data structures common or nearly common to two such software systems can readily be communicated between them. E.g., we can easily communicate strings, real numbers, integers, and arrays of integers and/or reals between SETL, FORTRAN, APL, and SNOBOL, but cannot expect SNOBOL to digest the internal complications of SETL sets, or SETL to handle SNOBOL patterns. Nevertheless, the rudimentary data objects listed in the preceding sentence can be expected to have nearly identical representations in most language implementations; moreover, minor discrepancies in the representation of these relatively standard objects, e.g. array headers differing in detail, can easily be compensated for by small routines belonging to the software interface between languages.

Confining our attention therefore to rudimentary data objects, what we want is for a procedure written in any of the languages of a hybrid system to be able to call procedures written in any of the other languages, passing and receiving simple data objects with common representations. We shall also

make the simplifying assumption that every one of our intercommunicating software systems operates in its own memory subarea, and that all these subareas can be loaded together in a common (virtual) memory.

Our aim is to define a simple semantic primitive which can support the necessary intersystem linkages without either making any restrictive assumptions about the manner in which the individual systems have been implemented or requiring any type of system support not apt to be present in most general operating systems. Even though a fair range of complications need to be faced, we can outline a scheme which, suitably implemented, would suffice to interconnect a wide variety of separately developed software environments. To this end, we can proceed as follows. In each of the software subsystems $S_1, S_2, \ldots$ that is to be linked, provide a standardized 'interface' or 'gate' procedure

(*) $\qquad GATE(N, R, X_1, Y_1, X_2, Y_2, \ldots)$,

either of a variable number N of parameters, if the subsystem in question allows this, or of some large fixed number K of parameters of which only N will be used on any particular call. Accordingly, when procedure P_1 belonging to the software subsystem S_1 wishes to call a procedure P_2 belonging to a different software subsystem S_2 , we can proceed as follows.

(i) P_1 can call the $GATE_1$ available to it, indicating the number N of parameters that are being passed, and passing these parameters as $Y_1, Y_2, \ldots$. The parameter R is intended to identify both the routine P_2 that is being called and the subsystem S_2 to which P_2 belongs. The auxiliary parameters $X_1, X_2, \ldots$ serve to identify the type and size of the corresponding parameters $Y_1, Y_2, \ldots$ (if these types and sizes are not directly indicated in the data objects $Y_1, Y_2, \ldots$ themselves, as, e.g., they would not be if S_1 is a FORTRAN software domain) and to control the manner of transmission and conversion from the values of $Y_1, Y_2, \ldots$ available on the 'sending' side S_1 to the corresponding values seen on the 'receiving' side S_2. The following forms of conversion will be typical.

(i) In the very simplest case, no nontrivial conversion is required: the address (or value) of a given Y_j can be passed directly to the 'receiving' side S_2.

(ii) A value Y_j may carry a descriptive header on the 'sending' side S_1 which is not needed on the 'receiving' side S_2. In such a case, it may be sufficient to add to the address of Y_j an offset which bypasses this header, and pass the adjusted address to S_2.

(iii) Conversely, the value Y_j may lack a header H which is necessary on the 'receiving' side S_2. In this case, it may be sufficient for the receiving $GATE_2$ procedure to build H in a detached location which $GATE_2$ can allocate, putting a pointer to the actual position of Y_j in H and then passing along a pointer to the position of H.

(iv) If either the value of Y_j is shared on the 'sending' side S_1 and S_2 expects arguments to be transmitted by reference and modified in place, or if S_2 expects Y_j to begin with a header not present on the S_1 side, it will be necessary for the 'sending' side $GATE_1$ to copy Y_j into an area Y_j' which $GATE_1$ is able to allocate, pass the copy Y_j' to the 'receiving' side $GATE_2$, and then copy or return the address of Y_j' after it has been modified by the 'receiving' side as the result of the inter-system call from S_1 to S_2.

Note that the parameters $(N,R,X_1,Y_1,...)$ of the $GATE_1$ call can be set up by an auxiliary routine R_1 available on the calling side S_1, to which only those parameters $Y_1,...Y_m$ that are actually used need be passed. This routine is, in effect, a sending-side representative of the receiving-side routine R_2 which is to be called. Of course, the code of R_1 must reflect knowledge of the parameter conversions required, as determined by the semantics of both the S_1 and the S_2 languages; however, R_1 can be written in the S_1 language alone.

When first received within S_2, a call via $GATE_2$ to a routine RR belonging to the subsystem S_2 can be passed along to an auxiliary 'receiver' routine P_2, having a standard name known to $GATE_2$, but otherwise written entirely in the S_2 language. P_2 can examine the second, procedure-designating parameter R of the $GATE_2$ call (see (*) above), and using it can determine the identity of RR. Then, after RR returns to P, P can call $GATE_1$ again, but in a manner that indicates that a return from a prior call to $GATE_2$, rather than a new call originating within S_2, is required. These conventions require a GATE to handle four types of calls:

(i.a) calls originating in the same software subsystem S as the GATE, and representing invocations of routines belonging to some other subsystem;

(i.b) calls originating in S, and representing returns to the subsystem from which S was called;

(ii.a) calls originating outside S, representing invocations of routines within S; and

(ii.b) calls originating outside S, representing returns to routines within S.

The GATE must distinguish between these four forms of calls, and handle them as follows. GATE calls of type (i.a) must trigger any appropriate sending-side conversions, possibly including value copying and the building of headers (or 'dope-vectors') as indicated above, together with recursive stacking of return addresses and of any parameters passed by value. GATE calls of type (ii.b) must trigger any necessary data reconversions, and must pop control and parameter return address from the stack on which they have been saved, also returning parameters that have been transmitted by value for delayed value return. GATE calls (i.b) return control, along with any parameter values transmitted by value, to the subsystem from which the call originated. GATE calls (ii.a) must stack the identifier of the subsystem in which the call originates, along with the addresses of parameters to be returned by value.

Note therefore that most of the work of interpreting the conversion control parameters X_j of a call (*) must be performed within the GATE routines themselves.

The intersystem linkage scheme which we have outlined requires only minor modification of the software systems which are to be linked. The minimum modification is to furnish each system S_j with a procedure having the list of parameters specified for a GATE, which simply passes these parameters along to an external procedure named $GATE_j$. The necessary GATE procedures can then be written separately in an appropriate low-level language, although, of course, their author must understand all relevant details of the manner in which the data items to be passed are stored in each of the separate systems to be linked, and also all the conversions and parameter-transmission actions to be triggered by allowed values of the parameters X_j of (*). An experiment to see how successfully such an isolated (if nontrivial) software package could be written (e.g. to link SNOBOL, SETL, FORTRAN, and LISP) would be worthwhile.

D. Software portability.

In the technological period now drawing to an end, it was often considered necessary to write large portions of software systems in assembly language in order to meet speed and memory constraints. Now, however, these constraints are relaxing, and software portability, which is radically sacrificed by the use of assembly language, is becoming considerably more important than the relatively marginal gains ordinarily achieved by use of assembly language. Several quite successful portable systems have by now been developed, and fairly consistent experience with these systems suggests the following overall conclusions.

(a) Portability is obtained by writing all other software in a portable language, which should ordinarily be either a relatively low-level 'systems implementation language' (something on the order of Ada, or of a severely restricted PL/I, or a modified PASCAL), or should be a *pseudo-assembly language*, i.e. an assembly language for a hypothetical machine which can easily be translated into the actual assembly language of a wide variety of existing machines.

(b) The first approach (use of a systems implementation language) seems preferable if a wide variety of compatible systems, rather than a single portable system, is to be created. If this approach is used, the compiler for the system implementation language should be written in the language itself, and should be provided with at least two kinds of code generator back ends. The first kind of code generator should produce some single, well-designed, highly transportable pseudo-assembly language code; the second class of code generator should produce true assembly language code, much more carefully tailored to the various machines to which the system will be ported. The reason why it is desirable to produce both pseudo-assembly code and true assembly code for various machines is as follows. Pseudo-assembly code can

generally be ported to a new machine by writing a simple, unoptimized translator which, given a machine M, maps each pseudo-assembly instruction into a corresponding code sequence for the machine M. If the pseudo-assembly language is well designed, this translator can normally be written and made operational in less than four man-weeks. Direct generation of code for a particular target machine will generally produce more efficient code, allowing execution speeds somewhat less than double those ordinarily achieved by translation of a pseudo-assembly language. However, development of a high-quality direct code generator, even for a systems-writing language designed with transportability in mind, is a more complex process, often requiring about six months. During this six-month development period it can be quite inconvenient to have to cross-compile repeatedly from another machine to the target machine M. It is much more convenient to make the transportable language system available on M as rapidly as possible, even at some modest loss of efficiency, and then to carry out the remainder of the development on M itself.

(c) If a system implementation language is used, the portable compiler provided for it should include an optimizer subphase capable of applying all standard global optimizations (e.g., redundant expression elimination, code motion, constant propagation, dead code elimination, and operator strength reduction) and of packing quantities into a parametrized collection of standard-length registers. This will make it possible to compile high quality code, often performing within a factor of two or better of assembly code, for a variety of machines.

(d) All other applications and software systems, including compilers and run-time systems for very-high-level languages, should then be written in the basic portable language. The components needed to implement a very-high-level language system will normally be as follows.

(i) Parsing and semantic analysis routines which analyze

very-high-level source text and transform it into intermediate text, which is essentially a sequence of instructions for an 'extended machine' in which all the fundamental operations of the very-high-level language are available as primitives. These routines do not differ significantly from those used in any other compiler; they can and should be written directly in the transportable systems-writing language.

(ii) Global optimization routines. These routines take the intermediate code sequences just mentioned and use information concerning the primitives occurring in these code sequences to deduce attributes of the data objects which these sequences will generate. Once these attributes are known, more efficient code sequences producing equivalent outputs can generally be deduced. The necessary optimization routines may differ in detail, but not in general flavor, from those used in connection with compilers for lower level languages, and should also be written directly in the transportable systems-writing language.

(iii) Code generators, which produce either directly interpretable symbolic instruction sequences, or true assembly language macro-sequences equivalent to these symbolic instruction sequences. These generators should also be written in the transportable systems language. Careful design will generally make it possible to use a single appropriately parametrized code generator to produce assembler macros for a variety of machines. It is therefore possible to make this component of the high-level language system transportable also, and largely machine-independent.

(iv) A support library, consisting of routines which implement the more complex operations of the very-high-level language. This library can also be written in the transportable systems language. To move the library to a new machine, it will only be necessary to redefine the field layouts within the low-level data structures used to represent the objects of the very-high-level language.

(v) Assembler macro-text defining each of the macros emitted by the code generators (cf. (iii) above). This final system component is the only one which is machine dependent. But this component is small, and should generally amount to no more than a very few thousand lines of assembler code.

To summarize, we can say that the approach outlined in the preceding paragraphs makes it possible to transport major high-level language systems, which may involve tens of thousands of lines, between machines without having to produce more than a few thousand lines of assembly code for each new machine to which the system is to be ported.

Bibliography

A general view of the programming process having many points of contact with that presented in this paper are found in:

M. Hammer [1979] *Application Oriented Software Research*, and also

M. Hammer & G. Ruth [1979] *Automating the Software Development Process*, both in: Research Directions in Software Technology (P. Wegner, ed.) MIT Press, Cambridge, Mass.

The use of mathematical notations and operations as the basis for a programming language has been explored by the designers of SETL, APL, and more recently in PROLOG and the CIP transformable language being developed at the Technical University of Munich. See:

R.B.K. Dewar, A. Grand, S-C Liu, E. Schonberg, and J. T. Schwartz [1979] *Programming by Refinement, as Exemplified by the SETL Representation Sublanguage* T.O.P.L.A.S., v. 1, pp. 27-49.

J. T. Schwartz [1973] *On Programming: An Interim Report on the SETL Project. Installment. I: Generalities. Installment II: The SETL Language and Examples of its Use.* Lecture Notes, Computer Sci. Dept., New York Univ.

A similar range of questions is discussed and a variety of possible approaches are outlined in:

B. Liskov and V. Berzins [1979] *An Appraisal of Program Specifications*, in: Research Directions in Software Technology, (P. Wegner, ed.) MIT Press, Cambridge, Mass.

Recently, the possibility of using predicate logic directly as a very-high-level programming language has been suggested. See:

M. H. van Eamden and R. A. Kowalski [1976] *The Semantics of Predicate Logic as a Programming Language*. J. ACM v. 23, pp. 733-742.

R. A. Kowalski [1974] *Predicate Logic as a Programming Language*. Proc. IFIP Congress 74, North Holland Publ. Co., Amsterdam, pp. 569-574.

R. A. Kowalski [1979] *Algorithm = Logic + Control*. C.A.C.M., v. 22, pp. 424-436.

F. Bauer, et al. [1977] *Notes on the Project CIP: Outline of a Transformation System*. Tech. Rept. TUM-7729, Tech. Univ. München, Inst. für Informatik.

F. Bauer, et al. [1978] *Towards a Wide Spectrum Language to Support Program Specification and Program Development*, SIGPLAN Notices, v. 13, no. 12.

Transformational derivation of programs has been studied by many authors during the past few years. See in particular:

R. Paige [1979] *Expression Continuity and the Formal Differentiation of Algorithms*. Thesis, New York University.

R. M. Burstall and J. Darlington [1976] *A System which Automatically Improves Programs*. Acta Informatica, v. 6, pp. 41-60.

R. M. Burstall and J. Darlington [1977] *A Transformation System for Developing Recursive Programs*. J.A.C.M., v. 24, pp. 44-67.

D. F. Kibler, et al. [1977] *Program Manipulation via an Efficient Production System*, SIGPLAN Notices, v. 12.

D. B. Loveman [1977] *Program Improvement by Source to Source Transformation*. J.A.C.M., v. 24, pp. 121-145.

T. Standish, et al. [1976] *The Irvine Program Transformation Catalogue*. Tech. Rept., Dept. of Information and Computer Science, U. C. Irvine.

M. A. Auslander and M. R. Strong [1978] *Systematic Recursion Removal*. C.A.C.M., v. 21, pp. 127-134.

H. Partsch & P. Pepper [1976] *A Family of Rules for Recursion Removal Related to the 'Tower of Hanoi' Problem*. Tech. Rept. 7612, Techn. Univ. München, Inst. für Informatik.

B. Wegbreit [1976] *Goal-directed Program Transformation.* IEEE Trans. on Software Engineering, v. SE-2, pp. 69-80.

S. Gerhart [1975] *Correctness-preserving Program Transformations.* Tech. Rept. CS-1975-4, Duke Univ., Durham, N.C.

(See also the previously cited work of Bauer et al.)

Various interesting (though fragmentary) attempts at automatic construction of programs will be found in:

P. D. Summers [1977] *A Methodology for LISP Program Construction from Examples.* J.A.C.M. v. 24, pp. 161-175.

M. Sintzoff [1978] *Inventing Program Construction Rules,* in: Proc. IFIP Conf. on Constructing Quality Software, North Holland Publ.

N. Dershowitz and Z. Manna [1977] *The Evolution of Programs: Automatic Program Modification.* I.E.E.E. Trans. on Software Engineering, v. SE-3, pp. 377-385.

M. Sharir [1980] *Algorithm Construction by Formal Differentiation,* to appear in: Mathematics and Computers with Applications.

A brief survey of the major techniques for formal proof of program correctness and an assessment of the potential for progress in this field is found in:

J. Schwartz [1979] *A Survey of Program Proof Technology,* Tech. Rept. 1, Dept. of Computer Science, Courant Inst. Math. Sci., New York Univ.

Facilities for user definition of object types related to, but not quite as abstract as those sketched in Section 2.c of the present report, are provided in the Algol 68, CLU, ALPHARD, and Ada languages. See later references for Ada, also:

A. van Wijngaarden, et al. [1975] *Revised Report on the Algorithmic Language Algol 68*, Acta Informatica, v. 5, pp. 1-236.

B. H. Liskov and S. N. Zilles [1974] *Programming with Abstract Data Types*. SIGPLAN Notices, v. 9, no. 4, pp. 50-59.

B. Liskov, and S. Zilles [1975] *Specification Techniques for Data Abstractions*. IEEE Trans. Software Engineering, v. 1, pp. 7-19.

B. Liskov, A. Snyder, R. Atkinson [1977] *Abstraction Mechanisms in CLU*. CACM, v. 20, pp. 564-576.

W. Wulf, R. L. London, and M. Shaw [1976] *An Introduction to the Construction and Verification of ALPHARD Programs*. IEEE Trans. Software Eng., v. 2, pp. 253-265.

The literature on 'external' or 'user-oriented' programming issues is voluminous, but largely impressionistic.

A brief but broad survey of application-oriented programming languages is found in the Programming Languages chapter of the recent COSERS report:

B. Arden, ed. [1980] *COSERS: The Computer Science and Engineering Research Study*, v. I, II. MIT Press, Cambridge, Mass.

The following papers describe two unusual programming languages with particularly strong applications orientations:

M. M. Zloof [1975] *Query-by-Example*. Proc. 1975 Nat. Computer Conf., AFIPS Press, Montvale, N. J., v. 44, pp. 431-438.

M. M. Zloof and S. P. de Yong [1977] *System for Business Automation* (SBA) *Programming Language*. C.A.C.M. v. 20, pp. 385-396.

An unusually perceptive account of user-oriented software design problems in the specialized area of numerical software is found in:

J. R. Rice [1979] *Software for Numerical Computation*, in: Research Directions in Software Methodology (P. Wegner, ed.) MIT Press, Cambridge, Mass.

An interesting proposal for a language emphasizing pseudoparallelism as a programming technique is found in :

C.A.R. Hoare [1978] *Communicating Sequential Processes*. C.A.C.M., v. 21, pp. 666-676.

A particularly perceptive account of other important pragmatic issues arising in externally motivated applications programs is found in :

G. J. Meyers [1976] *Software Reliability; Principles and Practices* John Wiley & Sons, Publ., New York.

See also:

J. Aron [1974] *The Program Development Process*. Addison Wesley Publ., Reading, Mass.

J. E. Bingham and G.W.P. Davies [1972] *A Handbook of Systems Analysis*. Halstead Publ., New York.

J. D. Dougar and R. W. Knapp [1974] *Systems Analysis Techniques*. John Wiley and Sons Publishers, New York.

J. Martin [1973] *Design of Man-Computer Dialogues*. Prentice-Hall Publ., Englewood Cliffs, New Jersey.

L. A. Belady and M. M. Lehman [1976] *A Model of Large Program Development.* IBM Systems Journal v. 15, pp. 225-252.

F. P. Brooks [1975] *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley Publ., Reading, Mass.

For another account of this area and a survey and extensive bibliography of various systems and approaches competitive with the suggestions advanced in the present paper, see:

B. Boehm [1979] *Software Engineering: R & D Trends and Defense Needs*, in: Research Directions in Software Technology (P. Wegner, ed.) MIT Press, Cambridge, Mass.; also

C. L. McGowan and R. C. McHenry [1979] *Software Management*, in: Research Directions in Software Technology (P. Wegner, ed.), pp. 207-253.

An interesting discussion of the pragmatically important program maintenance problem is found in:

L. Belady and M. Lehman [1971] *Programming System Dynamics, or the Metadynamics of Systems in Maintenance and Growth*, IBM Res. Rept. RC3546, Yorktown Heights, N.Y.

L. Belady and M. Lehman [1979] *The Characteristics of Large Systems*, in: Research Directions in Software Technology (P. Wegner, ed.) pp. 106-131.

Concurrent process languages have been studied very actively during the last five years. See:

P. Brinch-Hansen [1973] *Concurrent Programming Concepts*, ACM Computing Surveys, v. 5, pp. 223-245.

P. Brinch-Hansen [1975] *The Programming Language Concurrent PASCAL.* IEEE Transactions on Software Engineering, v. 1, pp. 199-207.

P. Brinch-Hansen [1977] *The Architecture of Concurrent Programs.* Prentice-Hall Publ., Englewood Cliffs, N.J.

C.A.R. Hoare [1974] *Monitors: An Operating System Structuring Concept.* C.A.C.M., v. 17, pp. 549-557.

N. Wirth [1977] *MODULA, a Language for Modular Multiprogramming,* Software Practice and Experience, v. 7, pp. 37-66.

P. Shaw [1978] *GYVE, a Programming Language for Protection and Control in a Concurrent Processing Environment, v. I, II.* Thesis, New York Univ.

Concurrent processing features have of course been incorporated in the new DoD Ada Language. See Chapter 9 of:

Preliminary Ada Reference Manual, Special Number of SIGPLAN Notices, v. 14, no. 6 [1979].

An interesting account of the special problems connected with data-base oriented operating systems is found in:

J. Gray [1978] *Notes on Data Base Operating Systems,* in: Operating Systems, an Advanced Course. Springer-Verlag, New York.

The recent literature contains numerous accounts of successfully transported software systems:

C. O. Grosse-Lindermann and H. H. Nagel [1976] *Postlude to a PASCAL-compiler Bootstrap on a DEC System/10.* Software Practice and Experience, v. 6, pp. 29-42.

O. Lecarme and M. C. Peysolle [1978] *Self-compiling Compilers: an Appraisal of their Implementation and Portability.* Software Practice and Experience, v. 8, pp. 149-170.

R. Miller [1978] *UNIX — a Portable Operating System?* Operating Systems Review, v. 12, pp. 32-37.

M. C. Newsey, P. C. Pode, and W. M. Waite [1972] *Abstract Machine Modeling to Produce Portable Software — A Review and Evaluation.* Software Practice and Experience, v. 2, pp. 107-136.

C. R. Snow [1978] *An Exercise in the Transportation of an Operating System.* Software Practice and Experience, v. 2, pp. 41-50.

T. Stuart [1976] *Adapting Large Systems to Small Machines.* SIGPLAN Notices, v. 11, pp. 144-150.

The SPITBOL compiler is a particularly successful transportable system; see:

R.B.K. Dewar and A. P. McCann [1977] *MACRO SPITBOL — A SNOBOL4 Compiler.* Software Practice and Experience, v. 7, pp. 95-113.

Our remarks on program testing cover just one aspect of the very important question of program testing.
For recent accounts of issues in this area, see:

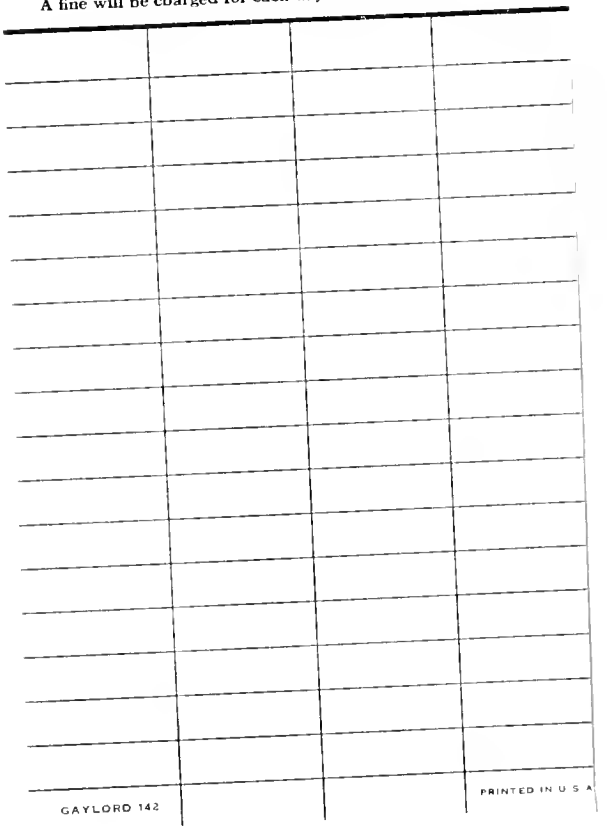
J. B. Goodenough [1979] *A Survey of Program Testing Issues,* in: Research Directions in Software Technology (P. Wegner, ed.) MIT Press, Cambridge, Mass., pp. 316-340

W. C. Hetzel (ed.) [1973] *Program Test Methods,* Prentice-Hall Publ., Englewood Cliffs, N. J.

R. de Millo, R. Lipton and F. Sayward [1978] *Program Mutation: a Method of Determining Test Data Adequacy,* Preprint, Yale Univ., New Haven, Conn.

Distribution List for Contract No. N00014-78-C-0639

Defense Documentation Center Cameron Station Alexandria, VA 22314	12 copies
Office of Naval Research Arlington, VA 22217	
Information Systems Program (437)	2 copies
Code 200	1 copy
Code 455	1 copy
Code 458	1 copy
Office of Naval Research Branch Office, Boston Bldg. 114, Section D 666 Summer Street Boston, MA 02210	1 copy
Office of Naval Research Branch Office, Chicago 536 South Clark Street Chicago, IL 60605	1 copy
Office of Naval Research Branch Office, Pasadena 1030 East Green Street Pasadena, CA 91106	1 copy
Naval Research Laboratory Technical Information Division, Code 2627 Washington, D. C. 20375	6 copies
Dr. A. L. Slafkosky Scientific Advisor Commandant of the Marine Corps (Code RD-1) Washington, D. C. 20380	1 copy
Naval Ocean Systems Center Advanced Software Technology Division Code 5200 San Diego, CA 92152	1 copy
Mr. E. H. Gleissner Naval Ship Research & Development Center Computation and Mathematics Department Bethesda, MD 20084	1 copy
Captain Grace M. Hopper (008) Naval Data Automation Command Washington Navy Yard Building 166 Washington, D. C. 20374	1 copy
Dr. Serafino Amoroso CENTACS Attn: DRDCO-TCS-BG Fort Monmouth, NJ 07703	1 copy
Defense Advanced Research Projects Agency Attn: Program Management/MIS 1400 Wilson Boulevard Arlington, VA 22209	3 copies
Director, National Security Agency Attn: R53, Mr. Glick Fort G. G. Meade, MD 20755	1 copy



NYU
Comp. Sci. Dept. c.2
TR-020
Schwartz
Internal, external, and ...

N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012